COLEGIO DE POSTGRADUADOS

INSTITUCIÓN DE ENSEÑANZA E INVESTIGACIÓN
EN CIENCIAS AGRÍCOLAS

CAMPUS MONTECILLO

POSGRADO EN SOCIOECONOMÍA, ESTADÍSTICA E
INFORMÁTICA
CÓMPUTO APLICADO

USO DE GPUS PARA EL AJUSTE DE REDES NEURONALES BAYESIANAS REGULARIZADAS

OCTAVIO RODRÍGUEZ MUÑOZ

T E S I S

PRESENTADA COMO REQUISITO PARCIAL PARA
OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS

MONTECILLO, TEXCOCO, ESTADO DE MÉXICO
2021

La presente tesis titulada: **Uso de GPUs para el ajuste de Redes Neuronales Bayesianas Regularizadas**, realizada por el alumno: **Octavio Rodríguez Muñoz**, bajo la dirección del Consejo Particular indicado ha sido aprobada por el mismo y aceptada como requisito parcial para obtener el grado de:

MAESTRO EN CIENCIAS

SOCIOECONOMÍA, ESTADÍSTICA E INFORMÁTICA
CÓMPUTO APLICADO

CONSEJO PARTICULAR

CONSEJERO Pérez Rdz.

Dr. Paulino Pérez Rodríguez

ASESOR

Dr. David Hebert del Valle Paniagua

ASESOR

Dr. Mario Alberto Vázquez Peña

Montecillo, Texcoco, Estado de México, 2021.

USO DE GPUS PARA EL AJUSTE DE REDES NEURONALES BAYESIANAS REGULARIZADAS

Octavio Rodríguez Muñoz, M.C.

Colegio de Postgraduados, 2021

RESUMEN

Una de las herramientas provenientes de la inteligencia artificial que más se utiliza en la actualidad para realizar análisis de datos son las redes neuronales artificiales (RNA). Estas surgen del intento de los investigadores de emular el modelo de aprendizaje del cerebro humano. A lo largo de los años, se han ido desarrollando técnicas que ayudan a optimizar las RNA, tanto en la parte matemática como computacional, esto último, valiéndose de los avances que han llevado a obtener procesadores más veloces y además con nuevas características como el multi núcleo y el multiproceso, mismas que hasta hace un par de décadas, sólo eran posibles en equipos de alto desempeño como supercomputadoras y clústers de cómputo paralelo.

La llegada de dicha potencia de cálculo a los equipos destinados al usuario común así como el desarrollo de tecnologías coadyuvantes para el cálculo de gráficos para el mercado de los video juegos (GPUs), han logrado que en la actualidad sea posible contar con una alta capacidad de paralelización en equipos tan accesibles como una computadora portatil o de escritorio. Es este último desarrollo tecnológico el que ha motivado el estudio de algoritmos de optimización que si bien, en muchos casos, ya habían sido paralelizados para equipos de alto rendimiento, requieren modificaciones para poder operar en los equipos que cuentan con GPUs para realizar los cálculos en paralelo. Sumado a lo anterior, se siguen explorando las capacidades de las GPUs para tratar algunos problemas de paralelización, sobre todo, cuando se trata de grandes cantidades de datos.

Este trabajo se enfoca en el ajuste de una red neuronal de una sola capa oculta aplicando regularización Bayesiana y cómputo paralelo, este último se realiza mediante el lenguaje y la arquitectura CUDA para GPUs. Para poder realizar algunos cálculos se hizo uso de bibliotecas como cuBLAS y CuSolver. Finalmente, se evaluó el desempeño del programa con dos implementaciones previas del algoritmo, para cómputo secuencial (biblioteca de funciones brnn del paquete estadístico R) y para cómputo paralelo mediante MPI.

Palabras clave: Cómputo paralelo, Redes Neuronales Artificiales, GPU, CUDA.

USE OF GPUS FOR FITTING BAYESIAN REGULARIZED NEURAL NETWORKS

Octavio Rodríguez Muñoz, M.C.

Colegio de Postgraduados, 2021

ABSTRACT

One of the tools from artificial intelligence that is most used today to perform data analysis is artificial neural networks (ANN). These arise from the attempt of researchers to emulate the learning model of the human brain. Over the years, techniques have been developed that help to optimize ANN, both in the mathematical and computational part, the latter, making use of the advances that have led to faster processors and also with new features such as multi core and multiprocessing, the same as until a couple of decades ago, were only possible on high-performance equipment such as supercomputers and parallel computing clusters.

The arrival of this computing power to the equipment intended for the common user as well as the development of auxiliary technologies for the calculation of graphics for the video games market (GPUs), have made it possible to have a high capacity today of parallelization in equipment as accessible as a laptop or desktop computer. It is this latest technological development that has motivated the study of optimization algorithms that, although, in many cases, had already been parallelized for high-performance equipment, require modifications to be able to operate on equipment that has GPUs to perform calculations in parallel. In addition to the above, the capabilities of GPUs are still being explored to deal with some parallelization problems, especially when it comes to large data sets.

This work focuses on the fitting process of a single hidden layer neural network applying Bayesian regularization and parallel computation, the latter is done using the CUDA language and architecture for GPUs. In order to perform some calculations, libraries such as cuBLAS and CuSolver were used. Finally, the performance of the program was compared with two previous implementations of the algorithm, for sequential computation (brnn library in R language) and for parallel computation using open MPI.

Key words: Parallel computing, Artificial Neural Networks, GPU, CUDA.

AGRADECIMIENTOS

Al Consejo Nacional de Ciencia y Tecnología (CONACyT) por el apoyo económico brindado para la realización de mis estudios.

Al Colegio de Postgraduados, por abrirme sus puertas y todo su apoyo para continuar con mi formación académica.

Al Dr. Paulino Pérez Rodríguez, por todo su apoyo, a pesar de las circunstancias, su infinita paciencia y la confianza para emprender este proyecto.

Al Dr. David Hebert del Valle Paniagua por todo su apoyo y consejos durante mis estudios.

Al Dr. Mario Alberto Vázquez Peña por su apoyo y colaboración desinteresada.

A mis profesores, por todo lo que aportaron a mi formación académica así como su disposición y paciencia para explicarme cuando me costaba entender algún tema.

A mis amigos y compañeros de generación, por todo su apoyo académico y moral en este proceso.

A mi familia, por todo su apoyo para poder llevar a buen puerto este proyecto.

DEDICATORIA:

A mis padres, por todo lo que me enseñaron, por esforzarse en hacer de mi una persona de bien, para ellos siempre dedicaré mis éxitos.

A mis hermanas, por siempre apoyarme y motivarme.

A mis sobrinos, por aligerar el cansancio con sus ocurrencias.

A la pequeña Kara, por desvelarse conmigo a lo largo de toda la maestría.

A mis amigos, los que ya estaban, los que llegaron, por ser siempre un apoyo.

CONTENIDO

LISTA DE CUADROS

LISTA DE FIGURAS

CAPÍTULO 1. INTRODUCCIÓN

La Inteligencia Artificial es la disciplina que se encarga de estudiar los procesos de aprendizaje que ocurren en los organismos biológicos y trata de reproducirlos para poder crear sistemas inteligentes. Dentro de esta disciplina hay dos enfoques, el primero, la Inteligencia Artificial Simbólica, se encarga de construir sistemas complejos que buscan aproximarse lo más posible al problema y diseñan una solución completa, como por ejemplo los sistemas expertos; el segundo, la Inteligencia Artificial sub simbólica, busca crear sistemas poco complejos los cuales se irán adaptando para poder desarrollar un sistema que pueda dar solución al problema, un ejemplo de este último son las redes neuronales artificiales (Isasi y Galván, 2004).

Los componentes de una red neuronal, es decir, las neuronas, se conectan a través de enlaces llamados sinapsis, a cada uno de estos enlaces se le asigna un valor llamado peso, si entre dos neuronas no hay conexión entonces el peso es cero. Es el conjunto de estos pesos lo que determina la salida de una red neuronal. Además, las neuronas se organizan en capas, el número de estas puede impactar de manera significativa en la complejidad de la red (Heaton, 2008).

El proceso para asignar los pesos a las conexiones entre neuronas se conoce como "entrenamiento de la red neuronal". Este proceso se repite hasta que en una prueba de validez de la red neuronal, el valor se encuentra dentro de un límite aceptable. El entrenamiento de las redes neuronales se puede dividir en tres grandes grupos: supervisado, no supervisado y métodos híbridos que combinan a los dos anteriores (Heaton, 2008).

El uso más común de las redes neuronales es para la resolución de problemas como: Clasificación, predicción, reconocimiento de patrones y optimización.

El presente proyecto busca evaluar la mejora en tiempos de cálculo de una red neuronal artificial con regularización Bayesiana mediante su implementación en la arquitectura CUDA para cómputo paralelo en unidades de procesamiento gráfico (GPUs por sus siglas en inglés). Para este fin se exploran las bases estadísticas que dan soporte al modelo de aprendizaje supervisado como son la regresión lineal y la regularización Bayesiana. Para la implementación del software se utilizará el algoritmo de Levenberg-Marquardt cuya aplicación ha sido revisada por varios autores como Foresse y Hagan (1997), Guzmán *et*

al. (2018) y Pérez-Rodríguez *et al.* (2013).

Además de la base estadística, el marco teórico explora las arquitecturas que permiten la paralelización de los procesos dentro de las computadoras y algunos ejemplos de la utilización de GPUs para realizar cómputo de propósito general en paralelo así como las bibliotecas que permitirán la realización de cálculos dentro de la GPU.

Finalmente, el software que resulte de la investigación se pondrá a prueba con distintos conjuntos de datos para poder comparar su desempeño con otras implementaciones del algoritmo.

1.1 Objetivo

General Implementar mediante el uso de bibliotecas especializadas de álgebra lineal las operaciones matriciales de inversion, factorización de Cholesky y la aproximación de la matriz Hessiana del algoritmo de ajuste de Redes Neuronales Bayesianas Regularizadas en el lenguaje de programación CUDA C que es parte de la plataforma de desarrollo de las unidades de procesamiento gráfico (GPUs) de NVIDIA.

Particular Comprobar la eficiencia y el desempeño de la aplicación desarrollada mediante el análisis de las predicciones y comparación del tiempo de ejecución con otras implementaciones del algoritmo de ajuste utilizado: R (para un solo procesador) y openMPI/C++ (para múltiples procesadores).

CAPÍTULO 2. REVISIÓN DE LITERATURA

2.1 Modelos de Regresión Lineal Múltiple

En estadística se cuenta con herramientas para modelar las relaciones entre variables, llamadas explicativas y valores observados en una escala continua, entre estas herramientas, una de uso común es la regresión lineal (Shalev-Shwartz y Ben-David, 2014).

Para quienes se dedican a la estadística y las ciencias de la computación, regresión se centra en la distribución de una variable de respuesta y que está condicionada (o depende) de uno o más covariables (o predictores) $x_1, \ldots, x_p$ (Berk, 2011).

En el modelo de regresión lineal múltiple, una variable de respuesta y puede ser predicha por múltiples (p) covariables ($x_1, \ldots, x_p$) (Heumann *et al.*, 2016). Esto hace que el modelo sea una extensión del modelo de regresión lineal simple, mismo que puede escribirse como sigue:

$$y = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p + e, \tag{2.1}$$

donde β_0 es el intercepto (el punto donde la recta de la ecuación corta el eje Y), $\beta_1, ..., \beta_p$ son los coeficientes de regresión, $x_1, ..., x_p$ las covariables y e es una variable aleatoria no observable con distribución normal con media cero y varianza σ^2.

El modelo en forma matricial se escribe como:

$$\boldsymbol{y} = \boldsymbol{X\beta} + \boldsymbol{e}, \tag{2.2}$$

donde

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad \boldsymbol{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1p} \\ 1 & x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & & & & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{np} \end{bmatrix}, \quad \boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix}, \quad \boldsymbol{e} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{bmatrix}, \tag{2.3}$$

donde $\boldsymbol{X}$ es una matriz de orden $n \times (p+1)$, n es el número de observaciones de cada covariable $x_1, \ldots, x_p$; $\boldsymbol{y}$ es el vector de $n \times 1$ de las n observaciones, $\boldsymbol{\beta}$ el vector de $(p+1) \times 1$ de los coeficientes de regresión asociados a las covariables y $\boldsymbol{e}$ el vector de $n \times 1$ de los errores que se supone se distribuyen como una variable aleatoria normal multivariada con media $\mathbf{0}$ y varianza $\sigma^2 \boldsymbol{I}$.

2.1.1 Estimación de coeficientes

En la práctica, los valores de $\beta_0, \beta_1, \ldots, \beta_p$ son desconocidos, así que se deben usar los datos existentes para estimar los coeficientes (James *et al.*, 2013), esto se puede obtener por medio del método de mínimos cuadrados, dicha estimación se consigue minimizando la suma de cuadrados del error, es decir:

$$Q(\beta_0, \ldots, \beta_p) = SCE = \sum_{i=1}^{n} e_i^2 = \mathbf{e}^t\mathbf{e} = (\mathbf{y} - \mathbf{x}_i^t\boldsymbol{\beta})^t(\mathbf{y} - \mathbf{x}_i^t\boldsymbol{\beta}). \tag{2.4}$$

Si la matriz $\boldsymbol{X}$ es de rango completo por columnas, es posible minimizar la SCE en (2.4) y se puede mostrar que el vector de parámetros de regresión se puede obtener como sigue:

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}^t\boldsymbol{X})^{-1}\boldsymbol{X}^t\boldsymbol{y}, \tag{2.5}$$

donde $\hat{\boldsymbol{\beta}}$ representa un vector que contiene las estimaciones de $\beta_0, \beta_1, \beta_2, \ldots, \beta_p$.

2.2 Regresión No Lineal

Existen muchos casos donde los fenómenos observados no se ajustan a modelos lineales, cuando esto ocurre, existe un modelo que permite que la variable dependiente y sea expresada en términos de cualquier función $f(\cdot)$ de variables independientes y parámetros. El modelo es el siguiente:

$$y_i = f(\boldsymbol{x}_i^t, \boldsymbol{\theta}) + e_i, \tag{2.6}$$

donde $f(\boldsymbol{x}_i^t, \boldsymbol{\theta})$ es una función no lineal de las variables independientes, $\boldsymbol{x}_i^t$, es un vector de p observaciones y $\boldsymbol{\theta}$ es un vector de p parámetros.

De acuerdo al teorema de Kolmogorov (Kolmogorov, 1957) "Cualquier función continua que toma valores en los reales $f(x_1, \ldots, x_p)$ definida en $[0,1]^n$, $n \geq 2$, puede ser representado en la forma:

$$f(x_1, \ldots, x_p) = \sum_{q=1}^{2p+1} g_q \left(\sum_{i=1}^{p} h_{iq}(x_i) \right), \tag{2.7}$$

donde $g_q(\cdot)$ y $h_{iq}(\cdot)$ son funciones de una variable y $h_{iq}(\cdot)$ son funciones monótonamente crecientes independientes de f" (Du y Swamy, 2014).

2.3 Factorización Cholesky

Dentro de la comunidad científica, existen 3 métodos de factorización ampliamente utilizados para resolver sistemas de ecuaciones de la forma: $\boldsymbol{Ax} = \boldsymbol{b}$, estos son QR, LU y Cholesky (Haidar *et al.*, 2016). En este tipo de sistema de ecuaciones $\boldsymbol{A}$ es una matriz de coeficientes conocidos, $\boldsymbol{x}$ es un vector de valores desconocidos (incógnitas) cuyo número de filas corresponde con el número de filas y columnas de $\boldsymbol{A}$ y $\boldsymbol{b}$ es un vector de constantes conocidas de las mismas dimensiones que el vector $\boldsymbol{x}$.

La factorización de Cholesky, también conocida como descomposición Cholesky, es principalmente usada en la solución numérica de sistemas de ecuaciones donde $\boldsymbol{A}$ es una matriz simétrica positiva definida (spd), es decir, su determinante es mayor que cero y tiene la forma $\boldsymbol{A} = \boldsymbol{LL}^t$ donde $\boldsymbol{L}$ es una matriz triangular inferior de $(n \times n)$ con los elementos positivos de su diagonal. Un uso común de este método es la inversión de matrices (Meyer y Tier, 2013).

La descomposición de Cholesky puede emplearse para resolver el sistema de ecuaciones lineales recién descrito re-escribiendo el sistema como sigue:

$$\boldsymbol{Ax} = \boldsymbol{LL}^t\boldsymbol{x} = \boldsymbol{b},$$

lo cual sugiere que $\boldsymbol{x}$ puede obtenerse en dos pasos:

1. Resolver $\boldsymbol{Lw} = \boldsymbol{b}$ para $\boldsymbol{w}$, con $\boldsymbol{L}^t\boldsymbol{x} = \boldsymbol{w}$ note que no es necesario invertir $\boldsymbol{L}$ ya que es triangular.
2. Resolver $\boldsymbol{L}^t\boldsymbol{x} = \boldsymbol{w}$ para $\boldsymbol{x}$.

Note que para resolver los sistemas en los pasos 1 y 2 se puede hacer mediante los algoritmos de substitución hacia adelante y sustitución hacia atrás y en ningún caso es necesario invertir la matriz $\boldsymbol{L}$.

Si se desea invertir la matriz $\boldsymbol{A}$ esto se puede hacer utilizando también la descomposición de Cholesky y existen dos procedimientos principales.

Procedimiento 1: Método de la matriz aumentada

Resolver los sistemas de ecuaciones $\boldsymbol{AX} = \boldsymbol{I}$, donde $\boldsymbol{X}$ es una matriz de incógnitas de dimensión $n \times n$ e $\boldsymbol{I}$ es la matriz identidad de dimensión $n \times n$. Los sistemas pueden ser re-escritos de la forma siguiente:

$$(\boldsymbol{LL}^t)\boldsymbol{X} = (\boldsymbol{LL}^t)[\boldsymbol{x}_1, ..., \boldsymbol{x}_n] = \boldsymbol{I},$$

lo cual sugiere el procedimiento siguiente para calcular la inversa:

1. Resolver $(\boldsymbol{LL}^t)\boldsymbol{x}_1 = \boldsymbol{i}_1$, para $\boldsymbol{x}_1$, donde $\boldsymbol{i}_1$ es la primera columna de la matriz identidad.
2. Resolver $(\boldsymbol{LL}^t)\boldsymbol{x}_2 = \boldsymbol{i}_2$, para $\boldsymbol{x}_2$, donde $\boldsymbol{i}_2$ es la segunda columna de la matriz identidad.

   $\vdots$

n. Resolver $(\boldsymbol{LL}^t)\boldsymbol{x}_n = \boldsymbol{i}_n$, para $\boldsymbol{x}_n$, donde $\boldsymbol{i}_n$ es la última columna (n) de la matriz identidad.

Al final del proceso se tendrá como solución $\boldsymbol{X} = [\boldsymbol{x}_1, ..., \boldsymbol{x}_n]$, la cual corresponde a la inversa de $\boldsymbol{A}$, para lo cual fue necesario resolver n sistemas de ecuaciones lineales de la forma $\boldsymbol{Ax} = \boldsymbol{b}$ descrita anteriormente.

Procedimiento 2: Método de la matriz aumentada con matrices triangulares

El algoritmo para la inversión de la matriz usando el método de Cholesky consta de 3 pasos (Benner *et al.*, 2013) que son los siguientes:

1. Se descompone la matriz spd en el producto $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^t$, donde la matriz inferior $\boldsymbol{L} \in \mathbb{R}^{n \times n}$ es el factor Cholesky.

2. El factor Cholesky es explícitamente invertido $\boldsymbol{L} \rightarrow \boldsymbol{L}^{-1} = \bar{\boldsymbol{U}}$. Esto se puede hacer utilizando nuevamente el método de la matriz aumentada, es decir se resuelven los sistemas:

$$\boldsymbol{L}\boldsymbol{X} = \boldsymbol{I},$$

   de donde se obtiene que $\boldsymbol{X} = \boldsymbol{L}^{-1}$.

3. La inversa es obtenida como $\boldsymbol{A}^{-1} = \bar{\boldsymbol{U}}\bar{\boldsymbol{U}}^t = \boldsymbol{U}^{-1}\boldsymbol{U}^{-1\,t}$.

2.4 Redes Neuronales Artificiales

2.4.1 Breve historia de las redes neuronales

El objetivo de las redes neuronales artificiales es producir máquinas de procesamiento paralelo que emulen una red neuronal biológica de la manera más fiel posible para la resolución de problemas que resultan complejos para las lógicas de cómputo convencionales (Isasi y Galván, 2004).

El inicio de la investigación en el campo de las redes neuronales puede ubicarse en 1938 cuando Rashevsky comenzó con sus estudios en neurodinámica, representando las funciones de las redes de neuronas como ecuaciones diferenciales (Mehrotra *et al.*, 1996). Es hasta 1943, que con su trabajo, McCulloch y Pitts demostraron que con bloques similares a neuronas podían realizar todas las operaciones lógicas (Fyfe, 1996). Este trabajo se considera como el primer modelo matemático de redes neuronales (Suzuki, 2013).

Para 1957, Frank Rosenblatt crea la máquina perceptrón, la cual es capaz de realizar clasificación de patrones simples (Fyfe, 1996). Esta máquina toma una suma ponderada de las entradas y envía como salida 1 si el resultado de la suma se encuentra por encima del límite de activación, es decir, realiza una discriminación lineal. Entre sus principales limitantes está el hecho de que sólo se puede tener una capa y que sólo puede ser aplicada a problemas en los cuales la separación lineal funciona adecuadamente (Suzuki, 2013).

El siguiente modelo que se presenta es ADALINE (Adaptive Linear Element), desarrollado por Widrow y Hoff (1960) el cual presenta pocas diferencias con perceptrón, la principal, su regla de aprendizaje llamada "regla delta", que minimiza el error entre la salida deseada y la obtenida hasta un punto deseado (Fausett, 1993; Isasi y Galván, 2004).

Después de estos avances, hubo una reducción en la investigación sobre el tema de redes neuronales artificiales a causa del libro de Minsky y Papert (1969) que afirma que los perceptrons no son computacionalmente universales; esta reducción duró al menos veinte años siendo hasta los años 1980s donde se retoma la investigación (Mehrotra *et al.*, 1996).

Es en esta década que se populariza el concepto de propagación hacia atrás (backpropagation en inglés), esto debido a los trabajos de McClelland y Rumelhart (1986) quienes retomaron las investigaciones previas de Werbos y John (1974) y Parker (1985) (Freeman y Skapura, 1991).

2.5 Redes Neuronales de una sola capa oculta

En las redes neuronales, el modelo más común es el de las redes neuronales de una sola capa oculta (SHLNN por sus siglas en inglés) (Pérez-Rodríguez *et al.*, 2013), dicho modelo puede ser planteado de manera matemática a partir del teorema de Kolmogorov (2.7), es decir:

$$y_i = \beta_0 + \sum_{k=1}^{s} w_k g_k \left(b_k + \sum_{j=1}^{p} x_{ij} \beta_j^{[k]} \right) + e_i, \quad i = 1, \ldots, n, \tag{2.8}$$

donde $(w_1, \ldots, w_s)^t$ son los pesos de la red; $(b_1, \ldots, b_s)^t$ son los llamados sesgos (interceptos) $(\beta_1^{[1]}, \ldots, \beta_p^{[1]}; \ \ldots; \ \beta_1^{[s]}, \ldots, \beta_p^{[s]})^t$ son los coeficientes de regresión, $\beta_j^{[k]}$ es un parámetro para una entrada j en la neurona $k = 1, \ldots, s)$ y $g_k(\cdot)$ es la función de activación (Pérez-Rodríguez *et al.*, 2013).

La función de activación mapea los datos de una línea real dentro de un intervalo abierto delimitado (-1,1). Un ejemplo de lo anterior es la función de tangente hiperbólica $(tanh(\cdot))$ dada por:

$$g_k(x) = \frac{1 - \exp(2x)}{1 + \exp(-2x)}. \tag{2.9}$$

La Figura 2.1 muestra una representación gráfica de una red neuronal de una sola capa oculta.

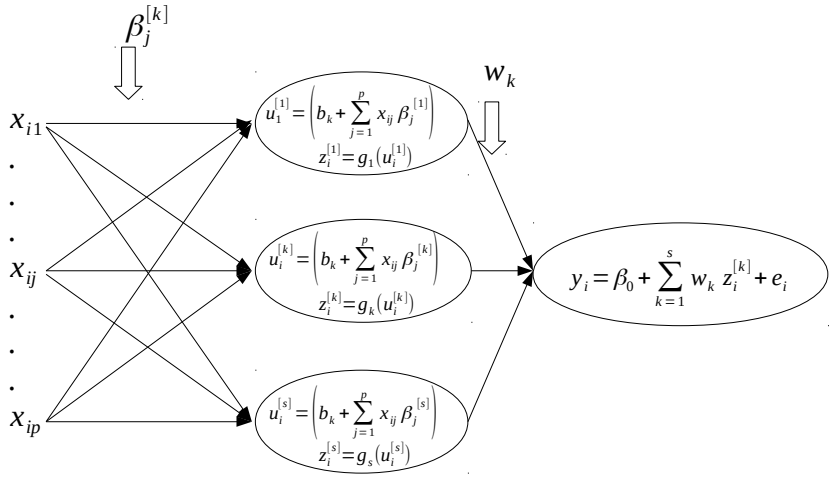

Figura 2.1: Representación esquemática de una red neuronal de una sola capa oculta. Fuente: Guzmán *et al.* (2018).

2.6 Redes Neuronales Regularizadas Bayesianas

2.6.1 Sobre ajuste

Uno de los principales problemas de las Redes Neuronales Artificiales de una sola capa es que, dada la flexibilidad del modelo no lineal, son susceptibles de sobre-ajustar los datos, esto es, el modelo predice de manera adecuada para el conjunto de datos de entrenamiento pero falla las predicciones sobre un nuevo conjunto de datos, es decir, tiene problemas de generalización. Esto suele ocurrir cuando aumenta el número de entradas y el de neuronas, lo que incrementa la cantidad de parámetros a estimar (Pérez-Rodríguez *et al.*, 2013). De acuerdo a Okut (2016) *"El sobre ajuste es inevitable si no se cuenta con algún tipo de regularización"*.

La regularización es el proceso de permitir sesgo en los parámetros llevando estos en la dirección de los valores más probables, esto reduce la varianza de las estimaciones pero agrega un costo, el sesgo. Visto de otro modo, es un compromiso entre la SCE y la suma de cuadrados de $\boldsymbol{\theta}$ a minimizar la función objetivo con respecto al espacio de los parámetros (peso y sesgo) (Okut, 2016).

2.6.2 Regularización Bayesiana

La Regularización Bayesiana (RB) es un método para lidiar con el problema de modelos sobre-parametrizados que se origina en modelos estadísticos que incluyen un número

masivo de variables explicatorias (p) pero pocas observaciones (n); en una combinación lineal junto con Redes Neuronales para determinar los parámetros óptimos de regularización, se obtiene lo que se conoce como Redes Neuronales Bayesianas Regularizadas (BRANN por sus siglas en inglés), en estos modelos implican la "imposición" de ciertas distribuciones "a priori" a los parámetros del modelo (Okut, 2016).

En general, en este tipo de regularización, los pesos y sesgos son funciones de parámetros desconocidos y, desde el punto de vista Bayesiano, son variables aleatorias (Gianola *et al.*, 2011; Du y Swamy, 2014), por lo que las redes pueden aprender la relación entre las variables explicatorias y las respuestas y así evitar plantearlas de forma arbitraria como ocurre en la regresión estándar. Este tipo de red neuronal también se puede ver como un modelo de regresión pero con el grado de no-linealidad dictado por el tipo de funciones de activación utilizadas (Gianola *et al.*, 2011).

El modelo descrito en el párrafo anterior se puede ilustrar de la siquiente manera: sea $\boldsymbol{\theta} = (w_1, \ldots, w_s;\ b_1, \ldots, b_s;\ \beta_1^{[1]}, \ldots, \beta_p^{[1]};\ \ldots;\ \beta_1^{[s]}, \ldots, \beta_p^{[s]}, \beta_0)^t$ siendo p el número de predictores y s el número de neuronas; estos son los parámetros de la red neuronal que se mencionan en (2.8) y se suponen desconocidos; suponga además que $\theta_j \sim N(0, \sigma^2_{\boldsymbol{\theta}})$. El proceso de ajuste de la red se trata de encontrar los valores de dichos parámetros, una forma de hacerlo es por medio de la minimización de la suma de cuadrados aumentada (Pérez-Rodríguez *et al.*, 2013), en la cual se agrega un término a la función objetivo, quedando de la siguiente forma:

$$F(\boldsymbol{\theta}) = \frac{1}{2\sigma_e^2}\sum_{i=1}^{n} e_i^2 + \frac{1}{2\sigma_{\boldsymbol{\theta}}^2}\sum_{j=1}^{m}\theta_j^2 = \beta E_D(\boldsymbol{\theta}) + \alpha E_{\boldsymbol{\theta}}(\boldsymbol{\theta}), \tag{2.10}$$

donde $E_D(\boldsymbol{\theta})$ es la suma de los errores residuales al cuadrado, $E_{\boldsymbol{\theta}}(\boldsymbol{\theta})$ es la suma de los parámetros (pesos y sesgos) de la red y

$$\beta = \frac{1}{2\sigma_e^2}, \alpha = \frac{1}{2\sigma_{\boldsymbol{\theta}}^2}. \tag{2.11}$$

Como puede verse, la función objetivo en este enfoque incluye la suma de los errores residuales al cuadrado y la suma de los pesos al cuadrado, lo que le ayuda a minimizar los errores de estimación y obtener buena capacidad de generalización (Kayri, 2016). Este método puede determinar los parámetros óptimos de regularización de una forma automatizada lo que elimina la necesidad de adivinar el tamaño óptimo de la red (Du y Swamy, 2014; Sun *et al.*, 2017).

Foresse y Hagan (1997) propusieron un método de pocos pasos que se repiten para resolver el problema de la estimación de coeficientes, dicho método está basado en la investigación

de MacKay (1992a) en la que se explora el uso del método de Bayes empírico para resolver problemas de regularización. El método propuesto ha sido retomado por diversos autores, entre ellos Gianola *et al.* (2011); Pérez-Rodríguez *et al.* (2013); Okut (2016); Guzmán *et al.* (2018). Los pasos (que se ilustran en la Figura 2.2) son los siguientes:

1. Inicializar α, β y el vector de parámetros a estimar $\boldsymbol{\theta}$.
2. Hacer un paso del algoritmo de Levenberg-Marquardt para minimizar la función objetivo $F(\boldsymbol{\theta}) = \beta E_D(\boldsymbol{\theta}) + \alpha E_{\boldsymbol{\theta}}(\boldsymbol{\theta})$.
3. Calcular el número efectivo de parámetros $\gamma = m - 2\alpha Traza(\boldsymbol{H})^{-1}$ haciendo uso de la aproximación a la matriz Hessiana disponible en el algoritmo de Levenberg-Marquardt: $\boldsymbol{H} = \bigtriangledown^2 F(\boldsymbol{\theta}) \approx 2\beta \boldsymbol{J}^t \boldsymbol{J} + 2\alpha \boldsymbol{I}_n$ donde $\boldsymbol{J}$ es la matriz Jacobiana de los errores del conjunto de entrenamiento y m es el número de elementos en $\boldsymbol{\theta}$.
4. Calcular nuevas estimaciones para los parámetros de la función objetivo: $\alpha_{new} = \frac{\gamma}{2E_{\boldsymbol{\theta}}(\boldsymbol{\theta})}$ y $\beta_{new} = \frac{n-\gamma}{2E_D(\boldsymbol{\theta})}$.
5. Repetir los pasos 2 a 4 hasta alcanzar la convergencia.

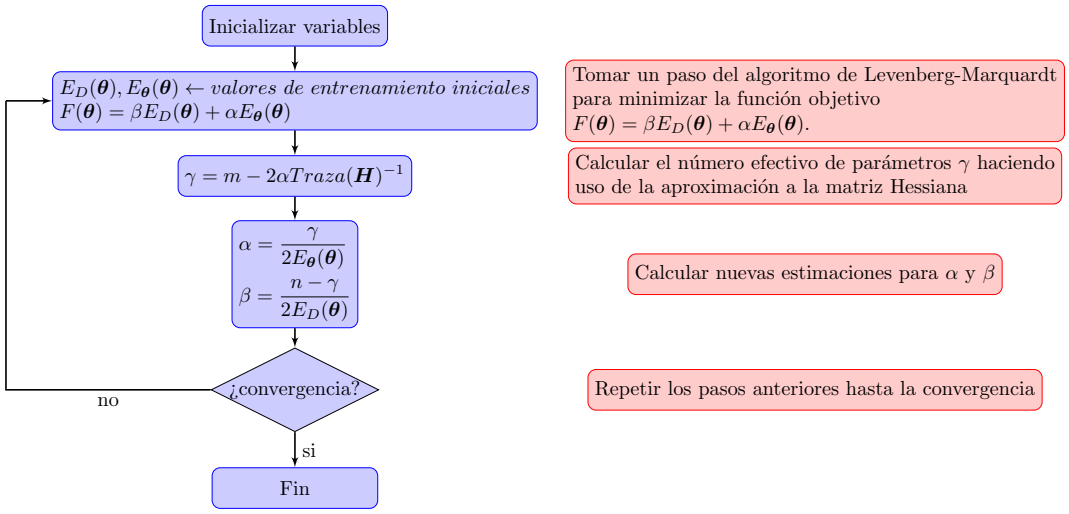

Figura 2.2: Diagrama de flujo del procedimiento de regularización Bayesiana.

2.6.3 Algoritmo de Levenberg-Marquardt

Es una variación del método de Newton diseñado para minimizar funciones que son sumas de cuadrados de otras funciones no lineales. Es muy útil para el entrenamiento de redes neuronales donde la función a minimizar corresponde a la media del error cuadrado (Demuth *et al.*, 2014).

De manera resumida, las iteraciones del algoritmo funcionan de la siguiente manera:

1. Presentar todas las entradas de la red y calcular los pesos correspondientes y los errores. Calcular la suma de cuadrados sobre todas las entradas, $F(\boldsymbol{x})$.
2. Calcular la matriz Jacobiana. Calcular las sensitividades con las relaciones recurrentes, después de inicializar. Aumentar las matrices individuales en las sensitividades de Marquardt. Calcular los elementos de la matriz Jacobiana.
3. Obtener Δx_k usando: $\Delta x_k = -[\boldsymbol{J}^t(\boldsymbol{x}_k)\boldsymbol{J}(\boldsymbol{x}_k) + \mu_k \boldsymbol{I}]^{-1}\boldsymbol{J}(\boldsymbol{x}_k)\boldsymbol{v}(\boldsymbol{x}_k)$.
4. Recalcular la suma de errores cuadrados usando $x_k + \Delta x_k$. Si esta nueva suma es más pequeña que la calculada en el paso 1, entonces dividir μ entre μ_{dec}, sea $x_{k+1} = x_k + \Delta x_k$ y regresar al paso 1. Si la suma de cuadrados no está reducida, entonces multiplicar μ por μ_{inc} y volver al paso 3.

Este algoritmo, como se menciona en párrafos anteriores, ha sido retomado por diversos autores para explorar la regularización Bayesiana, entre ellos Foresse y Hagan (1997); Gianola *et al.* (2011); Pérez-Rodríguez *et al.* (2013) y se ilustra en la Figura 2.3.

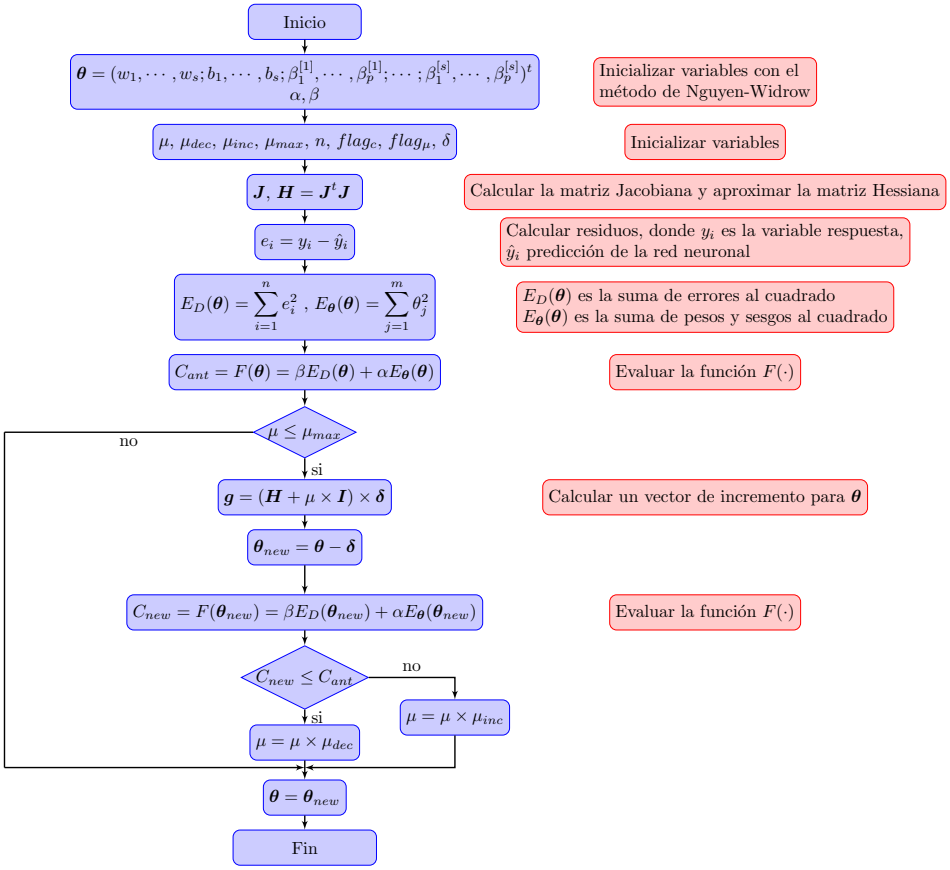

Figura 2.3: Diagrama de flujo del ajuste de la red neuronal.

2.7 Cómputo Paralelo

La mayoría de las computadoras están basadas en lo que se conoce como "Arquitectura Harvard" (Cheng *et al.*, 2014), la cual describe de manera conceptual el funcionamiento básico de un procesador. Los componentes principales de este modelo se pueden observar en la Figura 2.4 y son los siguientes:

- Memoria de instrucciones.
- Memoria de datos.
- Unidad de procesamiento central (Unidad Lógico-Aritmética y Unidad de Control).
- Interfaces de Entrada/Salida.

Se destaca en este modelo el hecho de que existen canales diferentes para los datos de programa y para los datos de instrucción, esto marca la diferencia con la arquitectura anterior, llamada Von Newman, la cual tenía un canal compartido para ambas secciones de la memoria (Hennessy y Patterson, 2019; Elahi, 2018).

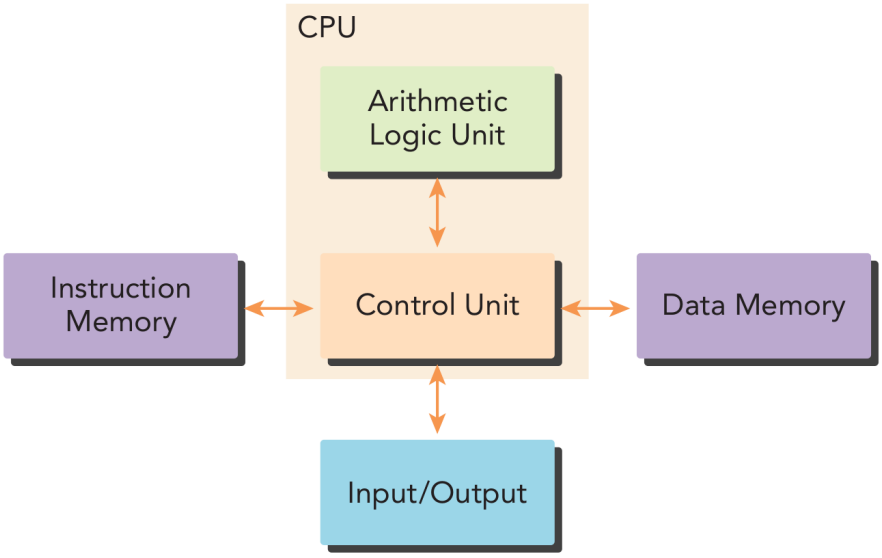

Figura 2.4: Arquitectura Harvard. Fuente: Cheng *et al.* (2014).

2.7.1 Arquitectura multinúcleo

Durante muchos años, las computadoras fueron vistas como máquinas de procesamiento secuencial, aún cuando internamente las cosas ocurrieran de otra manera, ya que, en

muchos casos, los procesadores tenían la capacidad de atender más de una instrucción y usan sistemas de interrupciones para lograrlo (Stallings, 2016).

A lo largo del tiempo, las mejoras en la capacidad de cómputo se centraban en aumentar la velocidad del reloj del procesador, así se podían realizar más operaciones en menos tiempo. Esta no es la única forma de aumentar el rendimiento pero sí la más común. Debido a las limitantes físicas en la fabricación de microchips, una alternativa para obtener mejor rendimiento es aumentar la cantidad de procesadores en una computadora, algo similar a lo que se usa en supercómputo o en cómputo de alto rendimiento (HPC por sus siglas en inglés) (Sanders y Kandrot, 2011).

El HPC generalmente se refiere al uso de múltiples procesadores (en una o varias computadoras) para realizar de manera concurrente tareas cuya complejidad impide (o dificulta) su realización en un solo procesador (o una sola computadora). HPC es un concepto que implica tanto una arquitectura de computadoras como plataformas, herramientas y un paradigma de programación para conseguir los resultados esperados (Cheng *et al.*, 2014).

Es entre 2005 y 2006 cuando comienza la era de los procesadores de más de un núcleo "multi core", con el lanzamiento a nivel comercial procesadores de 2 núcleos de cómputo. En la actualidad es un estándar tener procesadores de 2, 4 ó más núcleos aún en los equipos de bajo costo destinados a uso doméstico (Sanders y Kandrot, 2011; Bryant y O'Hallaron, 2016).

2.7.2 Arquitectura multihilo

Otra vía para incrementar la velocidad de procesamiento ha sido el desarrollo de procesadores que cuentan con múltiples hilos. Un hilo es una parte del programa que cuenta con una porción de código (la que se está ejecutando) y los datos necesarios para poder ejecutar dicho código (valores de variables y estructuras de datos) (Kirk y Hwu, 2017). Estos hilos son manejados de manera secuencial por el procesador, pero reducen el tiempo de espera ya que se trata de instrucciones pequeñas que no demandan tanto tiempo del mismo. Muchos procesadores multinúcleo poseen soporte para multihilo (Hennessy y Patterson, 2019).

Además del multihilo secuencial, existe otra variante, en la que múltiples procesadores simples ejecutan un grupo de hilos de instrucciones, estos grupos de procesadores simples se agrupan en un tipo especial de procesador llamado "multi procesador de flujo" (stream multiprocessor en inglés, denotado por las siglas SM) o "multiprocesadores de hilo" donde el "flujo" es un conjunto de datos como un vector o una matriz. Cada uno de estos multi procesadores es capaz de ejecutar múltiples instrucciones de manera concurrente dentro de un grupo más grande de estos. Este tipo de arquitectura está estrechamente relacionada con la de las GPUs (Fayez, 2011).

2.7.3 Paralelismo

Desde un punto de vista de programación, un problema se puede dividir en problemas más pequeños, esto facilita su solución, cada una de estas partes es vista como un proceso de cómputo, existen entradas (datos), procesamiento y salidas (datos), cuando cada una de estas partes (llamadas tareas) depende de las salidas de otra, se habla de dependencia de datos (Cheng *et al.*, 2014). Tomando en cuenta lo anterior, cuando las tareas no dependen de otras, se pueden ejecutar de manera concurrente, se habla entonces de un paralelismo de tarea, por otro lado, también existe un tipo de paralelismo basado en los datos, esto es, cuando hay una gran cantidad de datos por procesar y estos pueden dividirse para reducir el tiempo de cómputo (Kirk y Hwu, 2017).

Existen dos formas de abordar el paralelismo, ya sea con muchos núcleos o con muchos hilos de procesamiento. El paralelismo de tarea suele enfocarse más en la arquitectura multinúcleo ya que puede utilizar diferentes núcleos para las diferentes tareas, por otro lado, el paralelismo a nivel de datos utiliza la arquitectura multi hilo para dividir los grandes conjuntos de datos en porciones pequeñas de código que pueden repartirse el tiempo de procesamiento.

Los objetivos del cómputo paralelo son los siguientes:

- Resolver un problema en menos tiempo.
- Resolver grandes problemas en una cierta cantidad de tiempo.
- Obtener mejores soluciones para ciertos problemas en una cantidad de tiempo.

2.7.4 Clasificación de los sistemas de cómputo paralelo

Una de las clasificaciones más conocidas para las arquitecturas de cómputo paralelo de multiprocesador se basa en la disponibilidad de la memoria principal para dividirlos en Sistemas de Memoria Compartida y Sistemas de Memoria Distribuida que se muestran en las Figuras 2.5 y 2.6 respectivamente, también conocidos como Acceso Uniforme a la Memoria (UMA por sus siglas en inglés) y Acceso No Uniforme a la Memoria (NUMA) (Fayez, 2011).

2.8 Unidades de Procesamiento Gráfico

El incremento en la carga de trabajo que demandan los videojuegos ha propiciado que se desarrollen “co-procesadores” que ayuden en dicha tarea, estos dispositivos cuentan

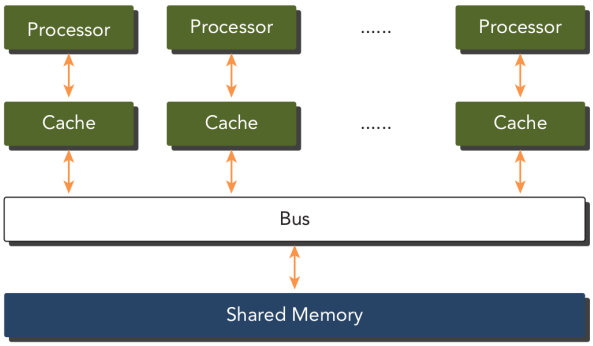

Figura 2.5: Memoria Compartida. Fuente: Cheng *et al.* (2014).

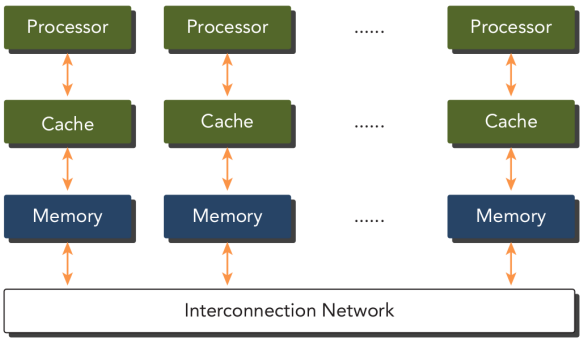

Figura 2.6: Memoria Distribuida. Fuente: Cheng *et al.* (2014).

con cientos de núcleos de procesamiento con capacidades bien definidas, esto provee al sistema de cómputo de una alta tasa de realización de operaciones de punto flotante (Meyer y Tier, 2013).

Las unidades de procesamiento gráfico (GPUs por sus siglas en inglés) son piezas de hardware que ayudan a la Unidad Central de Procesamiento (CPU por sus siglas en inglés) a acelerar algunos procesos, por este motivo, no es necesario que realicen todas las tareas del CPU. Así mismo, dado que fueron diseñadas para tareas en específico, algunas otras no las realizan de la mejor forma (Patterson y Hennessy, 2014). Una GPU es una máquina de alto grado de paralelización, lo que permite distribuir los procesos que realizar el sistema de cómputo (Sengupta *et al.*, 2007).

Es en la década de los 1990's, con el aumento en la demanda por aplicaciones con gráficos 3D, que surgen videojuegos para PC que empleaban ambientes gráficos más realistas, específicamente aquellos llamados de disparador en primera persona, los cuales impulsaron la adopción de gráficos 3D en computadoras personales. Es a partir de lo anterior que, compañías como NVIDIA, ATI Technologies y 3dfx Interactive, comienzan la comercialización y desarrollo de aceleradores gráficos (Sanders y Kandrot, 2011).

2.8.1 Uso de GPUs para propósito general

Originalmente, las GPUs fueron concebidas para el cálculo de colores de manera rápida, por lo que contaban con instrucciones de precisión simple y no estaban preparadas para realizar cálculos que no fueran de imágenes, esto cambió cuando algunos estudiantes y académicos se dieron cuenta de que podían utilizar la arquitectura de procesamiento de estos dispositivos para realizar cómputo paralelo mediante el "engaño" a la unidad al pasarle datos en forma de pixel para que realizara los cálculos. La anterior no era una tarea fácil ya que requería del dominio de un lenguaje orientado a gráficos (OpenGL) y el conocimiento de la arquitectura de la GPU, sin embargo, el incremento en este tipo de uso, que aumentó considerablemente la demanda de GPUs lo que hizo que las empresas fabricantes decidieran modificar el diseño para facilitar la realización de cómputo de propósito general en sus dispositivos (Soyata, 2018).

2.8.2 CUDA

En 2007, NVIDIA presentó su nuevo modelo de programación para GPUs junto con una nueva arquitectura desarrollada por la empresa: CUDA. Esta arquitectura fue pensada para realizar tanto cálculos gráficos como cálculos de propósito general (Kirk y Hwu, 2017).

La nueva arquitectura permitía el uso arbitrario de la memoria a todas las unidades de la GPU, sus unidades lógico-aritméticas (ALU por sus siglas en inglés) cumplían con el estándar de la IEEE para poder realizar cálculos de punto flotante y contaban con un conjunto de instrucciones para cómputo de propósito general en lugar de uno para cómputo gráfico (Vaidya, 2018).

Dada la complejidad para utilizar herramientas de desarrollo gráfico como OpenGL o DirectX para poder interactuar con los nuevos procesadores gráficos, para solucionar esto, NVIDIA desarrolló un lenguaje de programación a partir del C estándar añadiéndole nuevas funciones(Wilt, 2013). Este lenguaje, conocido como CUDA C, fue el primero diseñado para poder utilizar las GPUs para cómputo de propósito general. Junto con el lenguaje y un compilador para este, la empresa lanzó también un driver para poder explotar toda la capacidad de la GPU (Sanders y Kandrot, 2011).

2.8.3 Sistemas heterogéneos

Un sistema heterogéneo es aquel que combina distintos tipos de unidades de procesamiento, la versión más simple de estos es la que se puede encontrar en una computadora con un procesador multinúcleo que además tiene una GPU. Esta combinación le brinda al usuario la posibilidad de realizar cálculos que una

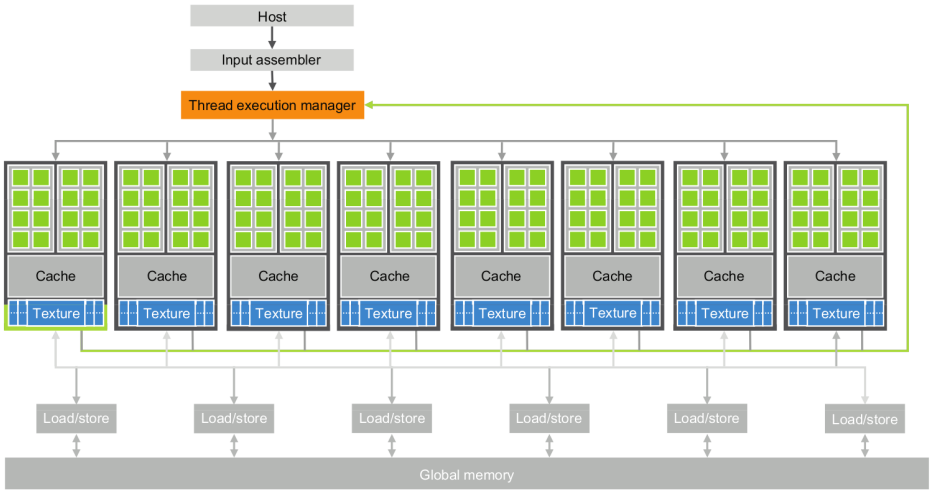

Figura 2.7: Arquitectura de una GPU. Fuente: Kirk y Hwu (2017).

computadora tradicional (con un solo procesador) no podría realizar. La Figura 2.8 muestra la combinación mencionada, la cual incluye además un sistema de comunicación entre las distintas unidades de procesamiento (Lukarski y Neytcheva, 2014).

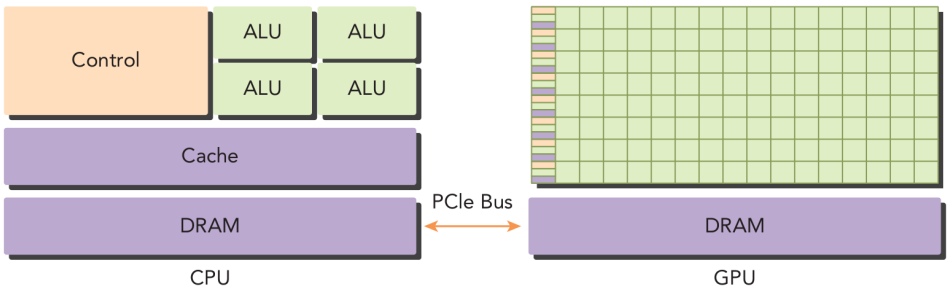

Figura 2.8: Arquitectura heterogénea CPU-GPU. Fuente: Cheng *et al.* (2014).

2.8.4 Bibliotecas para álgebra de matrices en CUDA

Antes del uso de GPUs ya era común el uso de bibliotecas especializadas para resolver problemas de álgebra lineal, tal es el caso de BLAS (Basic Linear Algebra Subprograms) y LAPACK (Linear Algebra PACKage) (Storti y Yurtoglu, 2016).

Para poder trabajar con diferentes tipos de operaciones en función de su intensidad de cómputo la biblioteca BLAS se divide en tres niveles de acuerdo a las operaciones que se van a efectuar (Soyata, 2018; Farber, 2011):

- **Nivel 1**, operaciones vector-vector.
- **Nivel 2**, operaciones matriz-vector.
- **Nivel 3**, operaciones matriz-matriz también conocidas como GEMM (General Matrix-Matrix Multiply).

Por otro lado, LAPACK es una biblioteca de alto nivel, basada en BLAS (realiza llamadas a las rutinas de esta biblioteca) para resolver problemas numéricos de álgebra lineal como sistemas de ecuaciones lineales, mínimos cuadrados, etc. (Anderson *et al.*, 1987).

cuBLAS

La biblioteca cuBLAS es una implementación de BLAS para la arquitectura CUDA que es provista por NVIDIA para resolver problemas de álgebra de matrices y vectores dentro de sus GPUs (Soyata, 2018) optimizada para matrices densas (Cheng *et al.*, 2014). El nivel 3, al igual que BLAS, realiza operaciones matriz-matriz las cuales se dividen por el tipo de datos que contienen dichas matrices, estos tipos de datos son:

- SGEMM, multiplicaciones matriz-matriz con datos de precisión simple.
- DGEMM, multiplicaciones matriz-matriz con datos de doble precisión.
- CGEMM, multiplicaciones matriz-matriz con datos complejos de precisión simple.
- ZGEMM, multiplicaciones matriz-matriz con datos complejos de doble precisión.

La biblioteca cuBLAS tiene tres tipos de implementación:

- La API de cuBLAS, que comienza con CUDA 6.0. dedicada a operaciones en una sola GPU.
- La API cuBLASXt, que comienza en CUDA 6.0. para operaciones con más de una GPU.
- La API cuBLASLt, que comienza en CUDA 10.1. para realizar operaciones con un uso eficiente de la memoria.

Para usar la API cuBLAS, la aplicación debe reservar memoria para las matrices y vectores requeridos en el espacio de memoria de la GPU, cargar los datos desde la memoria de la computadora al CPU, ejecutar la secuencia deseada de funciones cuBLAS y entonces

cargar los datos desde el espacio de memoria de la GPU de regreso a la memoria de la computadora.

Para usar la API cuBLASXt, la aplicación debe tener los datos en el anfitrión (la computadora) o en cualquiera de los dispositivos involucrados en el cómputo, y la biblioteca se hará cargo de repartir la operación y transferir los datos a una o varias GPUs en el sistema, dependiendo de lo que requiera el usuario.

cuBLASLt es una biblioteca ligera dedicada a las operaciones de multiplicación de matrices (GEMM) con una API flexible. Esta biblioteca añade flexibilidad en los diseños de los datos de la matriz, tipos de entrada, tipos de cómputo y también en elegir las implementaciones algorítmicas y heurísticas a través de la programabilidad de parámetros.

Para este proyecto se utilizará la API cuBLAS en su nivel 3 para las operaciones matriz-matriz y cuBLAS nivel 2 para las operaciones matriz-vector.

cuSOLVER

Este paquete de alto nivel es provisto por NVIDIA para tener rutinas del tipo LAPACKs como factorización de matrices y rutinas de solución triangular de matrices densas, así como rutinas para la solución de mínimos cuadrados de matrices dispersas y de eigen valores. También cuenta con una biblioteca para la re factorización (Storti y Yurtoglu, 2016; Ploskas y Samaras, 2016).

Las características anteriores se agrupan en tres componentes principales

- cuSolverDn, para la factorización de matrices densas y la soluciones, Cholesky, LU, QR, SVD y LDL^t así como permutaciones de matrices y vectores.
- cuSolverSP para las rutinas de matrices dispersas por ejemplo mínimos cuadrados.
- cuSolverRF para acelerar la re factorización.

Particularmente, cuSolverDn realiza la factorización Cholesky y devuelve una matriz triangular superior o inferior (Li *et al.*, 2019).

Para este proyecto haremos uso de cuSolverDn para las operaciones matriciales, como la descomposición Cholesky.

CAPÍTULO 3. RETOS COMPUTACIONALES PARA EL AJUSTE DE UNA RNA CON REGULARIZACIÓN

Tal y como lo señala Guzmán *et al.* (2018), las operaciones matriciales que forman parte del algoritmo de estimación representan un "cuello de botella" para el flujo del mismo. Para lidiar con esto se han propuesto estrategias, como el uso de bibliotecas especializadas, como las rutinas incluidas en las bibliotecas BLAS y LAPACK (Meyer y Tier, 2013; Benner *et al.*, 2013) que hacen más eficiente este cálculo y que además facilitan el proceso de programación. En el caso particular de este trabajo se planteó el uso de cuBLAS y cuSolver que son implementaciones para CUDA de BLAS y LAPACK respectivamente.

Los "cuellos de botella" se señalan en la Figura 3.1, como puede observarse, se trata de:

- El cálculo de aproximación de la matriz Hessiana ($\boldsymbol{H}$) mediante el producto de $\boldsymbol{J}\boldsymbol{J}^t$.
- El cálculo del vector $\boldsymbol{\delta}$ de incrementos para $\boldsymbol{\theta}$ mediante la solución del sistema de ecuaciones $\boldsymbol{g} = (\boldsymbol{H} + \mu\boldsymbol{I}) \times \boldsymbol{\delta}$.
- El cálculo de la inversa de la matriz $\boldsymbol{H}$, la cual se utiliza para calcular el número efectivo de parámetros γ.

3.1 Aproximación de la matriz Hessiana

Como se mencionó en la sección anterior, se busca aproximar la matriz Hessiana ($\boldsymbol{H}$) mediante el producto de la matriz Jacobiana ($\boldsymbol{J}$) mediante la ecuación:

$$\boldsymbol{H} = \boldsymbol{J}\boldsymbol{J}^t. \tag{3.1}$$

Una forma de tratar con matrices de gran tamaño es mediante el particionamiento por bloques o sub-matrices (Aggarwal, 2020), las cuales se pueden denotar de la forma:

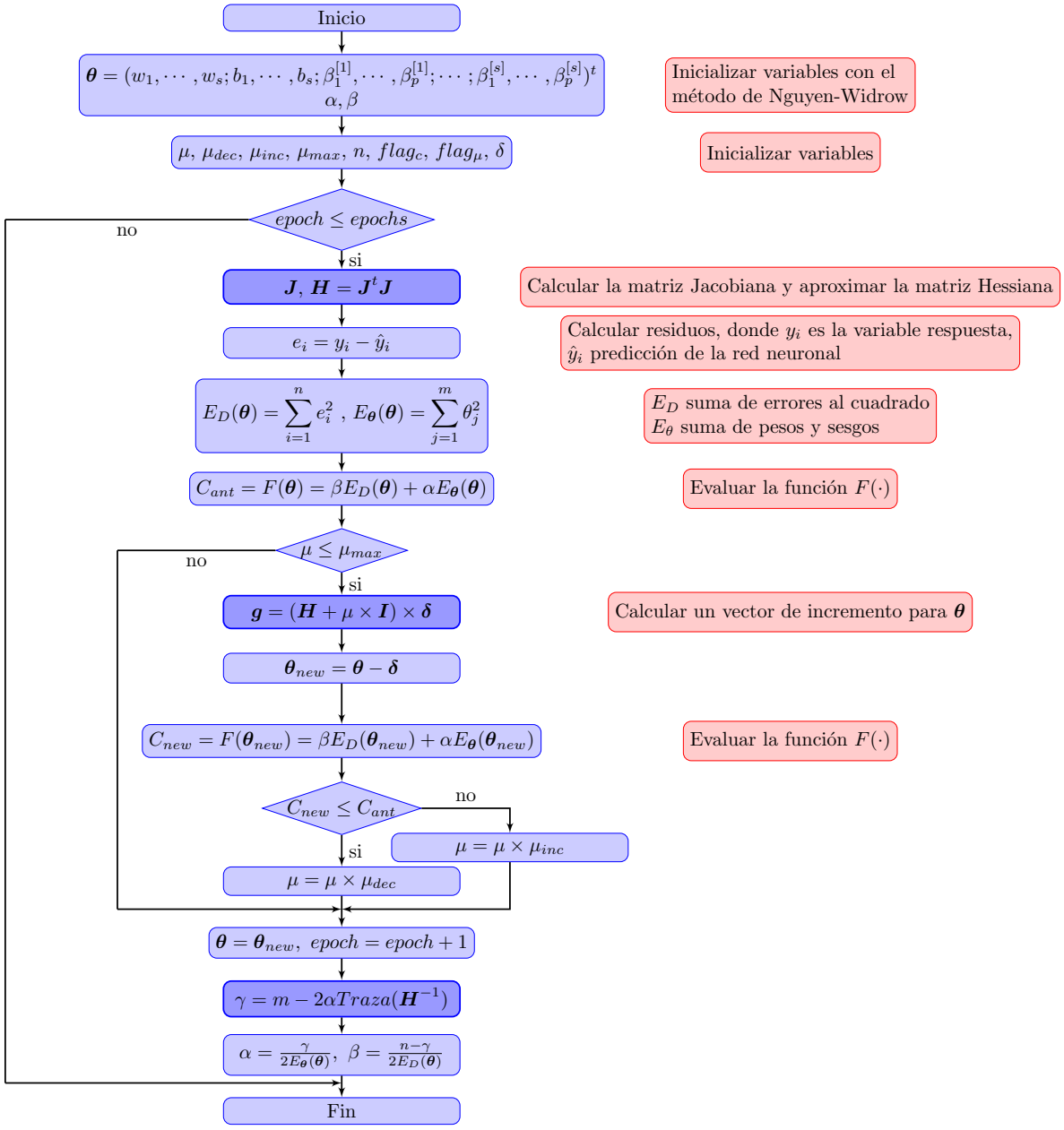

Figura 3.1: Diagrama de flujo del algoritmo de ajuste de una RNA, se resaltan los 3 pasos que se pueden paralelizar.

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} \\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22} \end{bmatrix}, \tag{3.2}$$

donde $\boldsymbol{A}_{11}$ y $\boldsymbol{A}_{12}$ así como $\boldsymbol{A}_{21}$ y $\boldsymbol{A}_{22}$ tienen el mismo número de renglones, además, tanto $\boldsymbol{A}_{11}$ y $\boldsymbol{A}_{21}$ como $\boldsymbol{A}_{12}$ y $\boldsymbol{A}_{22}$ tienen el mismo número de columnas. Es decir $\boldsymbol{A}$ o sus sub-matrices son de $m \times n$ y $\boldsymbol{B}$ o sus sub-matrices son de $n \times m$. El orden para el cálculo

del producto de dos matrices es de $O(n^3)$ (Gentle, 2017).

Para realizar una multiplicación de dos matrices $\boldsymbol{AB} = \boldsymbol{C}$, se puede ver como la multiplicación de varios vectores $\boldsymbol{b}_i$ por la matriz $\boldsymbol{A}$ (Gentle, 2017), o bien, se puede realizar a través de sub-matrices, en ambos casos estas deben ser conformables, es decir, que el número de columnas de la matriz $\boldsymbol{A}$ o de la sub-matriz $\boldsymbol{A}_{11}$ es igual al número de renglones de la matriz $\boldsymbol{B}$ o de la sub-matriz $\boldsymbol{B}_{11}$. En el caso de la multiplicación por bloques, esta debe verse de la siguiente manera:

$$\begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} \\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22} \\ \boldsymbol{A}_{31} & \boldsymbol{A}_{32} \end{bmatrix} \begin{bmatrix} \boldsymbol{B}_{11} & \boldsymbol{B}_{12} \\ \boldsymbol{B}_{21} & \boldsymbol{B}_{22} \end{bmatrix} = \begin{bmatrix} \boldsymbol{A}_{11}\boldsymbol{B}_{11} + \boldsymbol{A}_{12}\boldsymbol{B}_{21} & \boldsymbol{A}_{11}\boldsymbol{B}_{12} + \boldsymbol{A}_{12}\boldsymbol{B}_{22} \\ \boldsymbol{A}_{21}\boldsymbol{B}_{11} + \boldsymbol{A}_{22}\boldsymbol{B}_{21} & \boldsymbol{A}_{21}\boldsymbol{B}_{12} + \boldsymbol{A}_{22}\boldsymbol{B}_{22} \\ \boldsymbol{A}_{31}\boldsymbol{B}_{11} + \boldsymbol{A}_{32}\boldsymbol{B}_{21} & \boldsymbol{A}_{31}\boldsymbol{B}_{12} + \boldsymbol{A}_{32}\boldsymbol{B}_{22} \end{bmatrix}. \tag{3.3}$$

Así como las matrices involucradas en la multiplicación se pueden particionar en bloques, el producto de la operación $\boldsymbol{C}$ también puede particionarse, quedando expresados los operandos y el producto de la siguiente forma:

$$\underset{p_1 \quad p_s}{\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A}_{11} \dots \boldsymbol{A}_{1s} \\ \vdots \quad \vdots \\ \boldsymbol{A}_{q1} \dots \boldsymbol{A}_{qs} \end{bmatrix}} \begin{matrix} m_1 \\ \\ m_q \end{matrix} \quad , \quad \underset{n_1 \quad n_r}{\boldsymbol{B} = \begin{bmatrix} \boldsymbol{B}_{11} \dots \boldsymbol{B}_{1r} \\ \vdots \quad \vdots \\ \boldsymbol{B}_{s1} \dots \boldsymbol{B}_{sr} \end{bmatrix}} \begin{matrix} p_1 \\ \\ p_s \end{matrix} \quad , \quad \underset{n_1 \quad n_r}{\boldsymbol{C} = \begin{bmatrix} \boldsymbol{C}_{11} \dots \boldsymbol{B}_{1r} \\ \vdots \quad \vdots \\ \boldsymbol{C}_{q1} \dots \boldsymbol{B}_{qr} \end{bmatrix}} \begin{matrix} m_1 \\ \\ m_q \end{matrix} \quad , \tag{3.4}$$

entonces, para $\alpha = 1 : q$ y $\beta = 1 : r$ tenemos $\boldsymbol{C}_{\alpha\beta} = \sum_{\gamma=1}^{s} A_{\alpha\gamma} B_{\gamma\beta}$ (Golub y Van Loan, 2013).

3.2 Actualización del vector de incrementos $\boldsymbol{\delta}$ para $\boldsymbol{\theta}$

El vector de incrementos $\boldsymbol{\delta}$, el cual sirve para actualizar el vector de parámetros $\boldsymbol{\theta}$, es parte del sistema de ecuaciones lineales

$$\boldsymbol{g} = (\boldsymbol{H} + \mu \times \boldsymbol{I}) \times \boldsymbol{\delta}, \tag{3.5}$$

donde $\boldsymbol{g}$ es el gradiente de la red neuronal (Okut, 2016) definido por

$$\boldsymbol{g} = \boldsymbol{J'e}, \tag{3.6}$$

y $\boldsymbol{H}$ es la matriz Hessiana, μ es una constante e $\boldsymbol{I}$ es la matriz identidad. Los sistemas lineales se pueden expresar en forma matricial a partir de la ecuación

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \tag{3.7}$$

donde dados una matriz de datos $\boldsymbol{A}$ y un vector de datos $\boldsymbol{b}$ se busca encontrar a $\boldsymbol{x}$ (Neri, 2019), lo cual es uno de los problemas más comunes en ciencia e ingeniería (Zhang, 2020). Note que 3.5 se puede expresar en términos de 3.7, donde $\boldsymbol{A} := \boldsymbol{H} + \mu \times \boldsymbol{I}$, $\boldsymbol{b} = \boldsymbol{g}$ y $\boldsymbol{x} = \boldsymbol{\delta}$.

Una forma de resolver estos sistemas de ecuaciones es mediante la inversa de la matriz de datos $\boldsymbol{A}$, que, de acuerdo al teorema de Cramer, *"Si $\boldsymbol{A}$ es no singular, entonces existe una solución que satisface simultáneamente a todas las ecuaciones"* (Neri, 2019). Si $\boldsymbol{A}$ es una matriz no singular, es decir, su determinante es diferente de 0, entonces es invertible y existe una matriz $\boldsymbol{A}^{-1}$ y se cumple

$$\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{I}. \tag{3.8}$$

Dado lo anterior, es posible resolver el sistema de ecuaciones haciendo uso de la inversa de la matriz $\boldsymbol{A}$ de la forma:

$$\boldsymbol{A}^{-1}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}. \tag{3.9}$$

Este sistema de ecuaciones se puede solucionar aplicando la factorización de Cholesky para obtener $\boldsymbol{A}^{-1}$. En general, los pasos que se repiten para realizar la descomposición calcular los elementos de $\boldsymbol{L}$ son los siguientes (Aggarwal, 2020);

$$l_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2}; \quad j = 1, \ldots, n, \tag{3.10}$$

$$l_{ij} = \frac{(a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk})}{l_{jj}}; \; i = j+1, \ldots, n. \tag{3.11}$$

3.3 Cálculo de la inversa de la matriz Hessiana

Otra forma de encontrar la inversa de una matriz mediante la factorización de Cholesky es aplicar la división por bloques, para esto se descompone la matriz $\boldsymbol{A}$ en bloques así como la matriz $\boldsymbol{L}$ quedando de la siguiente manera:

$$\begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{21}^t \\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22} \end{bmatrix} = \begin{bmatrix} \boldsymbol{L}_{11} & 0 \\ \boldsymbol{L}_{21} & \boldsymbol{L}_{22} \end{bmatrix} \begin{bmatrix} \boldsymbol{L}_{11} & 0 \\ \boldsymbol{L}_{21} & \boldsymbol{L}_{22} \end{bmatrix}^t. \tag{3.12}$$

donde $\boldsymbol{A}_{11} \in \mathbb{R}^{r\times r}$, $\boldsymbol{A}_{22} \in \mathbb{R}^{n-r\times n-r}$, r es un factor de partición para los bloques y $\boldsymbol{L}$ es particionada de manera conformable (Golub y Van Loan, 2013). Al comparar los bloques se concluye:

$$\begin{aligned} \boldsymbol{A}_{11} &= \boldsymbol{L}_{11}\boldsymbol{L}_{11}^t, \\ \boldsymbol{A}_{21} &= \boldsymbol{L}_{21}\boldsymbol{L}_{11}, \\ \boldsymbol{A}_{22} &= \boldsymbol{L}_{21}\boldsymbol{L}_{21}^t + \boldsymbol{L}_{22}\boldsymbol{L}_{22}^t, \end{aligned}$$

lo cual sugiere tres pasos:

1. Calcular la factorización de Cholesky para $\boldsymbol{A}_{11}$ para obtener $\boldsymbol{L}_{11}$.
2. Resolver el sistema triangular para $\boldsymbol{L}_{21}$.
3. Calcular el factor Cholesky $\boldsymbol{L}_{22}$ de $\boldsymbol{A}_{22} - \boldsymbol{L}_{21}\boldsymbol{L}_{21}^t = \boldsymbol{A}_{22} - \boldsymbol{A}_{21}\boldsymbol{A}_{11}^{-1}\boldsymbol{A}_{21}^t$.

Existen varias formas de implementar estos pasos, ya sea de manera recursiva o no recursiva, en ambos casos, el orden de las operaciones es de $O(\frac{n^3}{3})$ a diferencia del tiempo normal para la descompocisión de Cholesky que es de $O(n^3)$ (Aggarwal, 2020).

CAPÍTULO 4. IMPLEMENTACIÓN DEL ALGORITMO PARA AJUSTE DE RNA CON REGULARIZACIÓN USANDO GPUS

4.1 Uso de las GPUs

El uso de GPUs para cómputo de propósito general persigue los siguientes objetivos:

- Maximizar la ejecución en paralelo.
- Minimizar el acceso a la memoria global.
- Mejorar el uso de los hilos.
- Mantener la alta intensidad aritmética de los cálculos.

Si alguno de estos objetivos no se cumple, el programa puede llegar a ser más lento que en el CPU (Ploskas y Samaras, 2016), ante lo cual se puede usar un sistema heterogéneo.

Los experimentos de Meyer y Tier (2013) indican que el uso de GPUs para realizar la inversa de matrices es una opción para acelerar los cálculos y además sugieren la escalabilidad de estos resultados con GPUs de mayor capacidad.

4.2 Propuesta de solución

Para el ajuste de la red neuronal de una sola capa oculta utilizando el enfoque Bayesiano se utilizan los algoritmos descritos en trabajos previos en los cuales se basa esta investigación, particularmente Foresse y Hagan (1997); Gianola *et al.* (2011); Pérez-Rodríguez *et al.* (2013) y Guzmán *et al.* (2018), siendo este último el que además aborda la paralelización del algoritmo para la solución del problema utilizando varios CPUs. Para resolver las operaciones matriciales de mayor costo computacional se hará

uso de las bibliotecas para álgebra lineal cuBLAS y cuSolver mismas que pueden utilizarse con CUDA.

El algoritmo propuesto por los autores citados es el del Levenberg-Marquardt, que busca minimizar la función $F(\boldsymbol{\theta})$ que es la suma de cuadrados del error aumentada. Dicho algoritmo utiliza operaciones matriciales que, de acuerdo a trabajos previos, representan un "cuello de botella" en la ejecución del programa para el entrenamiento de una red neuronal artificial regularizada Bayesiana.

Para realizar las operaciones matriciales, se propone el uso bibliotecas para GPUs de NVIDIA como cuBLAS y cuSOLVER.

- **Proceso 1. Aproximación de la matriz Hessiana mediante el producto de la matriz Jacobiana**. Al tratarse de una multiplicación de matrices se hace uso de la rutina `cublasDgemm` de la biblioteca cuBLAS, la cual, calcula de manera paralela la multiplicación de matrices densas. La rutina calcula el producto de una matriz $\boldsymbol{A}$ y una matriz $\boldsymbol{B}$, multiplica el resultado por un escalar α y suma a la matriz resultante el producto de la matriz $\boldsymbol{C}$ por un escalar β. La rutina es muy general y de forma simbólica las operaciones pueden representarse como sigue:

  $$\boldsymbol{C} = \alpha \times op(\boldsymbol{A}) \times op(\boldsymbol{B}) + \beta \boldsymbol{C},$$

  donde $op(\boldsymbol{X}) = \boldsymbol{X}$ o bien $op(\boldsymbol{X}) = \boldsymbol{X}^t$. El fragmento de código siguiente muestra como llamar la función, así como los argumentos de la misma, mismos que se describen detalladamente en el manual de cuBLAS.

```
1
2 cublasStatus_t cublasDgemm(cublasHandle_t handle,
3                            cublasOperation_t transa, cublasOperation_t transb,
4                            int m, int n, int k,
5                            const double *alpha,
6                            const double *A, int lda,
7                            const double *B, int ldb,
8                            const double *beta,
9                            double *C, int ldc)
```

- **Proceso 2. Cálculo del vector gradiente**. En este proceso se calcula $\boldsymbol{J'e} = \boldsymbol{g}$. Para realizar este producto se utiliza la función `cublasDgemv`. Las operaciones que realiza la función se pueden representar en forma simbólica como sigue:

  $$\boldsymbol{y} = \alpha \times op(\boldsymbol{A})\boldsymbol{x} + \boldsymbol{\beta} y,$$

  donde α, β son escalares, $op(\boldsymbol{A}) = \boldsymbol{A}$ o bien $op(\boldsymbol{A}) = \boldsymbol{A}^t$, $\boldsymbol{x}$ y $\boldsymbol{y}$ son vectores. El fragmento de código siguiente muestra como llamar la función, así como los argumentos de la misma, mismos que se describen detalladamente en el manual de cuBLAS.

```
1
2 cublasStatus_t cublasDgemv(cublasHandle_t handle, cublasOperation_t trans,
3                            int m, int n,
4                            const double *alpha,
5                            const double *A, int lda,
6                            const double *x, int incx,
7                            const double *beta,
8                            double *y, int incy)
```

- **Proceso 3. Solución del sistema de ecuaciones para calcular el incremento del vector $\boldsymbol{\theta}$.** Para este proceso de utilizan tanto cuBLAS como cuSolver. El sistema a resolver es:

$$(\boldsymbol{H} + \mu \times \boldsymbol{I})\boldsymbol{\delta} = \boldsymbol{g},$$

el cual puede ser re-escrito como:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b},$$

donde $\boldsymbol{A} = (\boldsymbol{H} + \mu \times \boldsymbol{I})$, $\boldsymbol{x} = \boldsymbol{\delta}$ y $\boldsymbol{b} = \boldsymbol{g}$. Para resolver el sistema, primero se obtiene la factorización de Cholesky de la matriz $\boldsymbol{A}$ para obtener $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^t$. La factorización se realiza empleando la rutina `cusolverDnDpotrf`, la cual sobre-escribe las entradas en la triangular superior o inferior de la matriz $\boldsymbol{A}$, según indique el usuario. El fragmento de código siguiente muestra como llamar la función, así como los argumentos de la misma, mismos que se describen detalladamente en el manual de cuSolver.

```
1
2 cusolverStatus_t
3 cusolverDnDpotrf(cusolverDnHandle_t handle,
4            cublasFillMode_t uplo,
5            int n,
6            double *A,
7            int lda,
8            double *Workspace,
9            int Lwork,
10           int *devInfo )
```

Una vez que se obtiene la matriz $\boldsymbol{L}$, se utiliza la rutina `cusolverDnDpotrs` para resolver el sistema de ecuaciones $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$. La rutina resuelve los sistemas de ecuaciones siguientes:

$$\boldsymbol{A}\boldsymbol{X} = \boldsymbol{B},$$

donde $\boldsymbol{A}$ es una matriz de coeficientes, $\boldsymbol{X}$ es una matriz de coeficientes desconocidos y $\boldsymbol{B}$ es una matriz de coeficientes conocidos. En nuestro caso $\boldsymbol{X} = [\boldsymbol{x}]$ y $\boldsymbol{B} = [\boldsymbol{b}]$. La rutina realiza sustitución hacia atrás y sustitución hacia adelante para resolver

los dos sistemas de ecuaciones triangulares resultantes. El código siguiente muestra la forma de llamar la función y los argumentos. Para mayores detalles ver el manual de cuSolver.

```
cusolverStatus_t
cusolverDnDpotrs(cusolverDnHandle_t handle,
           cublasFillMode_t uplo,
           int n,
           int nrhs,
           const double *A,
           int lda,
           double *B,
           int ldb,
           int *devInfo)
```

- **Proceso 4. Inversión de la matriz Hessiana para calcular el número efectivo de parámetros** γ. La matriz $\boldsymbol{H}$ se invierte primero realizando la factorización de Cholesky de la misma empleando la rutina `cusolverDnDpotrf` descrita anteriormente. Una vez que se obtiene la matriz $\boldsymbol{L}$ se utiliza la rutina `cusolverDnDpotrs` para resolver tantos sistemas de ecuaciones lineales como columnas de la matriz $\boldsymbol{H}$ para obtener la matriz $\boldsymbol{H}^{-1}$ utilizando el método de la matriz aumentada, es decir se resuelve el sistema:

$$\boldsymbol{AX} = \boldsymbol{I},$$

con $\boldsymbol{A} = \boldsymbol{H}$ y $\boldsymbol{B} = \boldsymbol{I}$. Una vez que se resuelve el problema se obtiene $\boldsymbol{X} = \boldsymbol{H}^{-1}$.

4.2.1 Equipo de cómputo y software

Para el desarrollo de la aplicación y la ejecución de todas las pruebas se utilizó un equipo de cómputo portátil (laptop) con las siguientes características:

- Procesador Intel Core(TM) i-7-8750H 2.20GHz con 6 núcleos físicos.
- 16gb de memoria DRAM DDR4.
- GPU NVIDIA GTX 1060 con 1280 núcleos CUDA y 3gb de RAM DDR5.
- Sistema Operativo GNU/Linux, distribución “Mint” versión 19.3.
- Kernel Linux 5.4.0-60.
- Versión del controlador (Driver) de NVIDIA 390.141.
- Versión del compilador CUDA 9.1.85.

El algoritmo para el ajuste de la red neuronal descrita en el trabajo fue codificado utilizando el lenguaje de programación C y para poder utilizar la GPU incluida en el equipo se utilizó CUDA. El código resultante se muestra en el anexo A, el cual incluye rutinas que se ejecutan en el CPU y en la GPU. La versión 2 de este código, la cual difiere de la original en que realiza el cálculo de la matriz Jacobiana dentro de la GPU para disminuir las transferencias de datos entre CPU y GPU, se muestra en el Anexo B. Las instrucciones de compilación se ejecutan desde la línea de comandos de Linux y se muestran en el Anexo C. Una vez que se compila y enlaza la aplicación se genera el ejecutable de la aplicación, cuyo uso se describe en la sección siguiente.

4.3 Ejecución de la aplicación

La aplicación desarrollada se ejecuta desde la línea de comandos (ejemplo Terminal en macOS o Linux, Shell en Windows). El comando para ejecutar la aplicación en la consola del sistema operativo Linux se detalla a continuación. En otros sistemas operativos el comando puede variar. La Figura 4.1 muestra a detalle cada una de los elementos que debe contener el comando para ejecutar la aplicación.

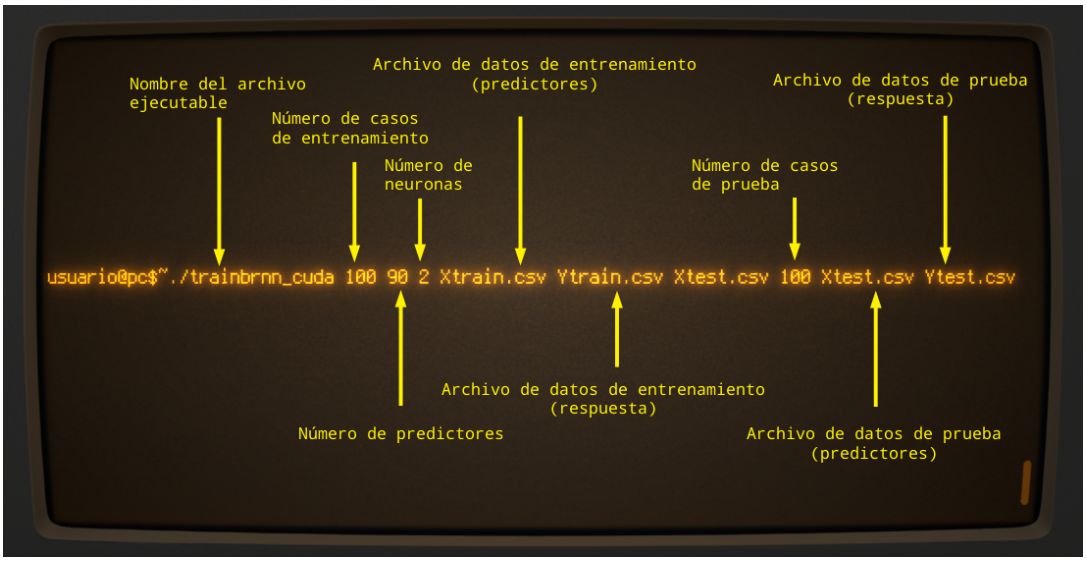

Figura 4.1: Comandos para utilizar la aplicación en la consola del sistema.

El software lee los datos de entrenamiento y prueba (opcional) desde archivos de texto ASCII separados por coma (csv=comma separated values, en inglés) los cuales pueden generarse fácilmente desde diversas aplicaciones de hoja de cálculo, como por ejemplo Microsoft Excel o Libre Office (https://www.libreoffice.org). En los archivos csv, las filas representan los casos y las columnas representan las variables y no tienen encabezado.

4.4 Comparación de tiempos de cómputo

En esta sección se compararon los tiempos de cómputo de 3 implementaciones del algoritmo para el ajuste de la red neuronal descrita, a saber: 1)La rutina `brnn` incluida en la biblioteca de funciones del mismo nombre (Pérez-Rodríguez y Gianola, 2020) en el paquete estadístico R (R Core Team, 2020), 2) Implementación del algoritmo en C/C++ usando las bibliotecas de mpi desarrollada por Guzmán *et al.* (2018) y 3) La implementación con C/CUDA en sus 2 versiones. La idea básica es medir el tiempo que toma hacer el ajuste de la red utilizando las tres implementaciones para diferente número de observaciones y neuronas.

Para la comparación se utilizan como datos de entrenamiento sub conjuntos de datos del archivo "YearPredictionMSD", el cual forma parte de "The Million Song Dataset" (Bertin-Mahieux *et al.*, 2011). Este conjunto de datos tiene como variable de respuesta el año en que fue lanzada una canción en un periodo que abarca de 1922 a 2011, para esto usa 90 covariables explicativas basadas en el "timbre" es cual es una característica obtenida de la API *Echonest* y que consiste de un vector de 12 dimensiones con valores centrados alrededor de 0 que representan un alto nivel de abstracción de la superficie espectral (Schindler y Rauber, 2014). Además el conjunto tiene más de 500,000 observaciones y no tiene elementos faltantes (Li *et al.*, 2016). Para el experimento se tomaron $n \in \{100, 200, 300, 400, 500, 600, 700, 800, 900, 1000\}$ registros del archivo de datos "YearPredictionMSD" y se ajustó la red neuronal con $s \in \{1, 2, 4\}$ neuronas. Cada ajuste se repitió 3 veces, con la finalidad de obtener tiempos promedio de ejecución de las aplicaciones. Para el caso específico de la implementación en mpi, los modelos se ajustaron utilizando 6 núcleos de cómputo arreglados en una rejilla de 3×2 núcleos. Para la aplicación desarrollada en CUDA se utilizaron 2 variantes, una que sólo utiliza CUDA en los cuellos de botella señalados y otra en la que se calcula la matriz Jacobiana en la GPU sin ningún tipo de paralelización pero disminuyendo el traslado de datos entre la GPU y el CPU. Por último, para la aplicación `brnn` del paquete estadístico R (R Core Team, 2020), se utilizó la configuración por defecto que es con 1 núcleo de procesamiento.

El Cuadro 4.1 presenta la comparación de los tiempos de cómputo para el ajuste de la red neuronal regularizada Bayesiana empleando las diferentes implementaciones del algoritmo para el ajuste. Los tiempos mostrados son el promedio de 3 ejecuciones para cada uno de los ejemplos. Puede observarse, el desempeño de la aplicación con GPU en la versión 1 no es el mejor en comparación con las otras implementaciones, también puede verse que la aplicación con GPU en su versión 2 tiene un desempeño similar a las otras 2 implementaciones pero el tiempo de cálculo aumenta conforme aumenta el número de neuronas (a 4) así como con el aumento del número de observaciones.

El comportamiento coincide por lo reportado por Meyer y Tier (2013), quienes mencionan que la ventaja de utilizar una GPU no es observable con conjuntos pequeños de datos, además, la caída en velocidad de la versión 2 de la implementación en CUDA demuestra

que, al no haber paralelización en el cálculo, todas las operaciones para obtener la matriz Jacobiana ocurren de manera serial por el tipo de procesamiento que se utiliza (sólo un núcleo CUDA), que no realiza ningún tipo de optimización. Por otro lado, la medición del tiempo que se realizó es del proceso completo, desde la lectura de datos hasta la obtención del resultado final y no sólo el tiempo de procesamiento de los cálculos matriciales.

Cabe destacar que la diferencia de velocidad entre las 2 versiones de la implementación en CUDA demuestran con claridad el costo de trasladar los datos entre CPU y GPU, sobre todo por el tamaño del conjunto de datos, que al tratarse de pocas observaciones, consume más tiempo en dichos traslados que en el cálculo, por lo que se confirma que la utilidad de una GPU se vuelve relevante cuando se trabaja con grandes cantidades de datos.

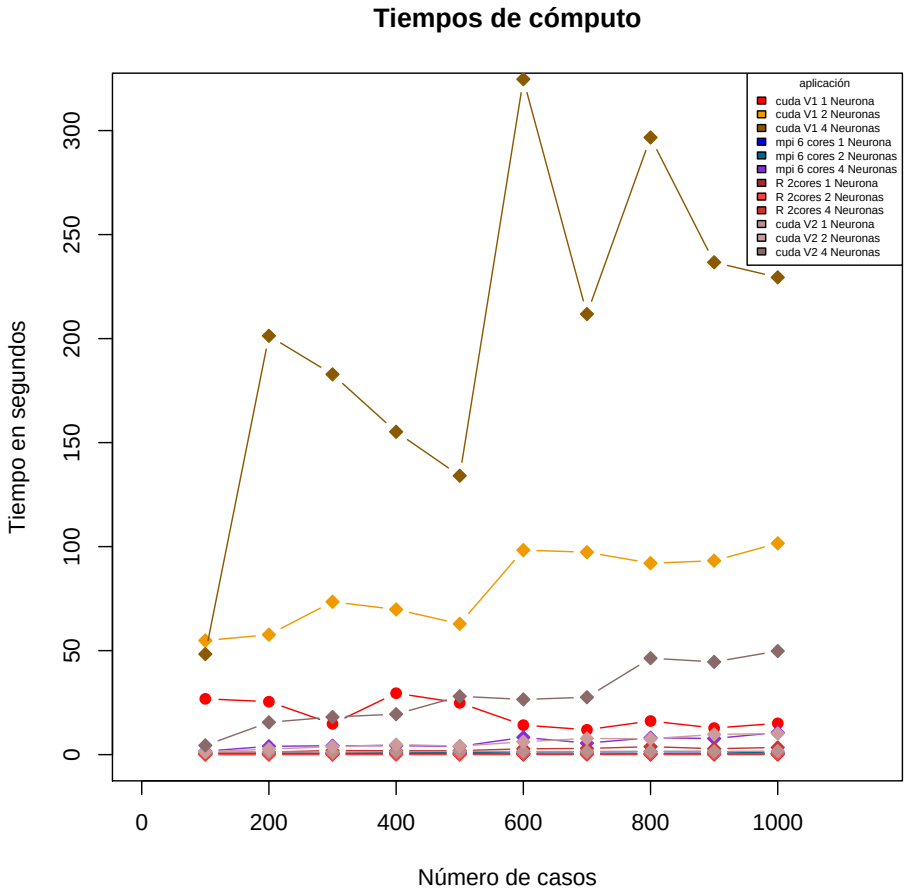

Figura 4.2: Relación tiempo/número de casos para cada una de las implementaciones con 1, 2 y 4 neuronas para cada una.

Cuadro 4.1: Comparación de tiempos de cómputo (en segundos) de cada una de las implementaciones del algoritmo para estimación.

	Conjunto de datos "YearPredictionMSD"		
Implementación	1 Neuronas	2 Neuronas	4 Neuronas
	(100 observaciones)		
brnn R	0.01633	0.02367	0.04733
mpi 6 cores	0.523333	0.756667	1.653333
CUDA V1	26.8300	54.8500	48.3300
CUDA V2	1.18333	1.34667	4.43667
	(200 observaciones)		
brann R	0.05733	0.05567	0.03300
mpi 6 cores	0.55667	0.83333	4.00000
CUDA V1	25.45667	57.653333	201.33000
CUDA V2	1.31333	2.66333	15.52667
	(300 observaciones)		
brann R	0.05467	0.03400	0.03800
mpi 6 cores	0.570000	0.956667	4.280000
CUDA V1	14.78667	73.47000	182.80667
CUDA V2	1.13667	3.82667	18.08333
	(400 observaciones)		
brann R	0.04300	0.09700	0.17800
mpi 6 cores	0.613333	1.236667	4.296667
CUDA V1	29.54000	69.77667	155.19667
CUDA V2	1.46333	4.84333	19.44667
	(500 observaciones)		
brann R	0.16933	0.24267	0.19733
mpi 6 cores	0.630000	0.976667	3.983333
CUDA V1	24.88000	62.87667	134.07000
CUDA V2	1.55667	4.08000	28.03667
	(600 observaciones)		
brann R	0.36033	0.40500	0.24367
mpi 6 cores	0.553333	1.356667	8.310000
CUDA V1	14.19000	98.31333	324.70333
CUDA V2	1.52333	6.19000	26.53000
	(700 observaciones)		
brann R	0.35467	0.38900	0.82467
mpi 6 cores	0.596667	1.446667	5.533333
CUDA V1	11.95333	97.31333	211.78000
CUDA V2	1.45000	7.77667	27.58333
	(800 observaciones)		
brann R	1.22333	1.99067	1.86133
mpi 6 cores	0.520000	1.453333	8.126667
CUDA V1	16.18000	92.03000	296.69333
CUDA V2	1.54667	7.61667	46.35667
	(900 observaciones)		
brann R	1.92300	2.78833	2.98067
mpi 6 cores	0.646667	1.530000	7.740000
CUDA V1	12.77667	93.21000	236.68000
CUDA V2	1.49000	9.72333	44.58000
	(1000 observaciones)		
brann R	3.81733	2.84633	3.44100
mpi 6 cores	0.610000	0.613333	10.636667
CUDA V1	15.0533	101.5800	229.4400
CUDA V2	1.68333	10.11000	49.79333

CAPÍTULO 5. APLICACIONES

En este capítulo se presentan dos ejemplos en los que se ajusta una red neuronal regularizada Bayesiana empleando el software desarrollado en CUDA. En el primer ejemplo se hizo uso del conjunto de datos "two input" incluida en la biblioteca brnn (Pérez-Rodríguez y Gianola, 2020) del paquete estadístico R (R Core Team, 2020). En el segundo ejemplo se utilizó el conjunto de datos "YearPredictionMSD", el cual se consultó a través del repositorio de la Universidad de California en Irvine.

5.1 Ejemplo1, 2 entradas 1 salida

En este primer ejemplo, se ajustó un conjunto de datos de dos entradas (x_1, x_2) y una salida (y) el cual es tomado de la biblioteca brnn que forma parte del paquete estadístico R. Estos datos fueron utilizados por Paciorek y Schervish (2003).

Se tomó como base el ejemplo incluido en la sección de ayuda de la función `brnn` de la biblioteca del mismo nombre. Este ejemplo se ajusta la red neuronal con 10 neuronas y función de activación tangente hiperbólica. El mismo modelo se ajusta utilizando el programa escrito en CUDA. Si bien podría parecer que 10 es un número arbitrario de neuronas, lo cierto que es al obtener el número efectivo de parámetros γ, el valor promedio de este es de 35.66 para la función `brnn` y de 36.93 para la aplicación en CUDA lo que muestra que 10 está cerca del valor óptimo para la cantidad de neuronas.

La Figura 5.1 muestra en detalle el diseño de la red neuronal artificial con 10 neuronas en la capa oculta a partir del diagrama de la Figura 2.1. Se puede ver como es procesado cada flujo de datos en todas las neuronas así como identificar los pesos y los sesgos de la red.

La Figura 5.2, muestra las predicciones del modelo y la gráfica de la función ajustada, en las Figuras 5.3 y 5.4 podemos ver las predicciones contra los datos observados para la aplicación en CUDA y la biblioteca brnn respectivamente; por último, en la Figura 5.5 se pueden observar las dos predicciones. Se puede ver que existe una relación lineal entre las predicciones como se muestra en las últimas gráficas.

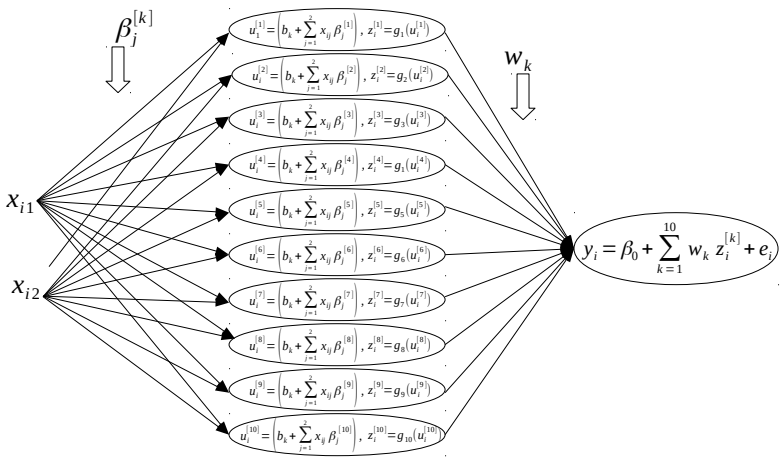

Figura 5.1: Diagrama de la red neuronal para el ejemplo de 2 entradas 1 salida.

5.2 Ejemplo2, Predicción del año

El segundo ejemplo se realiza utilizando el conjunto "YearPredictionMSD" que es un sub conjunto del conjunto de datos "The Million Song Dataset" (Bertin-Mahieux *et al.*, 2011), disponible para su uso en el repositorio para aprendizaje automático de la Universidad de California en Irvine (Dua y Graff, 2017). Este conjunto busca predecir el año en que una canción fue lanzada (entre 1922 y 2011) a partir de noventa covariables que son 12 sobre la media del timbre y 78 sobre la covarianza del timbre (Dua y Graff, 2017). Para este ejemplo se obtuvieron 10 sub conjuntos aleatorios del conjunto principal de entrenamiento, cada uno con mil (1000) elementos y sus respectivos sub conjuntos de prueba, cada uno con cien (100) elementos. Se ajustó la red neuronal regularizada Bayesiana con 1, 2, 4 y 8 neuronas y se obtuvieron las correlaciones de Pearson tanto para el conjunto de datos de entrenamiento y para la prueba, las cuales se muestran en el Cuadro 5.3, el cuadro se completa con las correlaciones obtenidas a partir de la regresión lineal.

Las Figuras 5.6 y 5.7 muestran la distribución de las correlaciones a partir del número de neuronas, los gráficos incluyen los resultados de las correlaciones obtenidas con la regresión lineal para mostrar la capacidad de generalización de la red neuronal artificial. Como puede observarse, la correlación es mayor en los 4 ejemplos de la red neuronal que la obtenida en la regresión lineal, además, puede observarse como aumenta dicha correlación conforme se incrementa la cantidad de neuronas en la red. Por otro lado, al observar los datos de prueba, puede notarse que la dispersión es mucho menor en los 4 ejemplos de la red neuronal a diferencia de la regresión lineal donde la dispersión es mayor.

Se probó mediante pruebas de Kruskal-Wallis (Hollander y Wolfe, 1999) y t de Student la diferencia de medias, de dichas pruebas se obtuvo que existen diferencias significativas

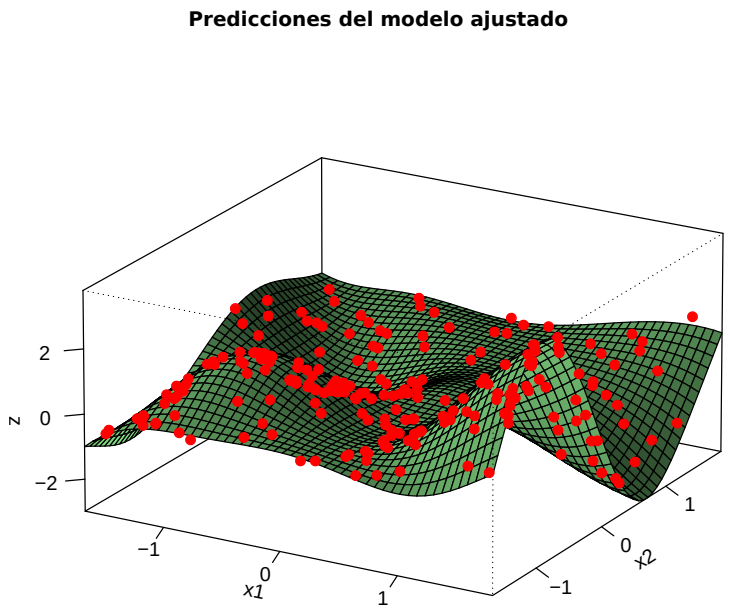

Figura 5.2: Ajuste de un modelo con 2 entradas y una salida realizado con el software desarrollado.

entre el poder predictivo (medido usando el coeficiente de correlación) de la red neuronal y la regresión lineal, para el caso del entrenamiento, desde la utilización de una sola neurona en la capa oculta, además se comprobó que existen diferencias significativas en la utilización de una o varias neuronas a partir de 4 neuronas. Para el caso de los datos de prueba, se encontró que las diferencias significativas entre la red neuronal y la regresión lineal se encuentran a partir de la utilización de 4 neuronas en la capa oculta; también se encontraron diferencias entre la utilización de una o varias neuronas. Los resultados de las pruebas entre cada una de las implementaciones de la red neuronal (cantidad de neuronas) y regresión lineal pueden consultarse en el Cuadro 5.1 para los datos de entrenamiento y en el Cuadro 5.2 para los datos de prueba.

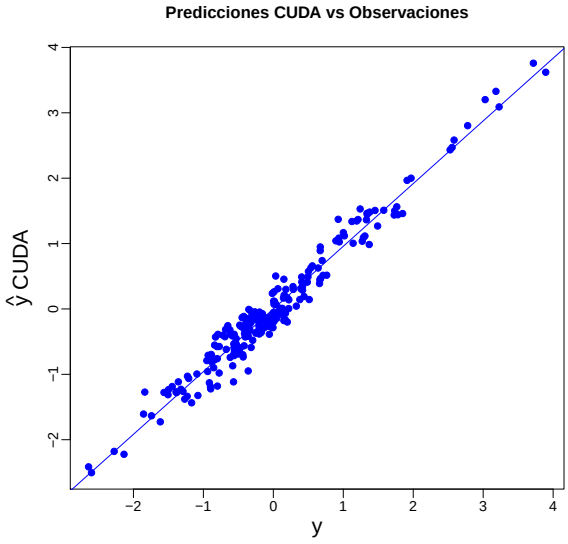

Figura 5.3: Relación entre los valores predichos y los valores observados (CUDA).

Cuadro 5.1: Resultados de la prueba t de Student de los datos de entrenamiento.

Implementaciones	Valor de t	p-value
n1 vs rl	4.6703	0.0003523
n2 vs rl	4.1797	0.0002961
n4 vs rl	23.425	$<$ 2.2e-16
n8 vs rl	24.503	$<$ 2.2e-16
n1 vs n2	-0.45423	0.6519
n1 vs n4	-26.048	$<$ 2.2e-16
n1 vs n8	-26.016	$<$ 2.2e-16
n2 vs n4	-19.6	$<$ 2.2e-16
n2 vs n8	-21.061	$<$ 2.2e-16
n4 vs n8	-3.9703	0.0002115

Cuadro 5.2: Resultados de la prueba t de Student de los datos de prueba.

Implementaciones	Valor de t	p-value
n1 vs rl	1.7885	0.1067
n2 vs rl	2.7375	0.02178
n4 vs rl	3.3974	0.007575
n8 vs rl	3.2594	0.009661
n1 vs n2	-4.9574	0.00001047
n1 vs n4	-10.454	1.144e-14
n1 vs n8	-11.479	$<$ 2.2e-16
n2 vs n4	-3.0387	0.003707
n2 vs n8	-2.5529	0.01438
n4 vs n8	1.015	0.3148

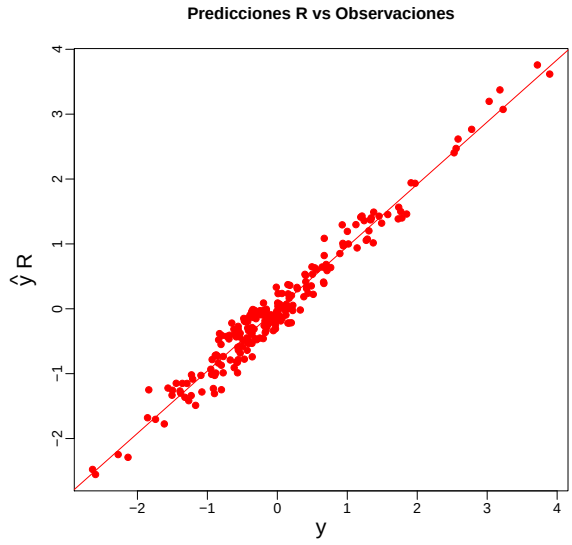

Figura 5.4: Relación entre los valores predichos y los valores observados (biblioteca brnn del paquete estadístico R).

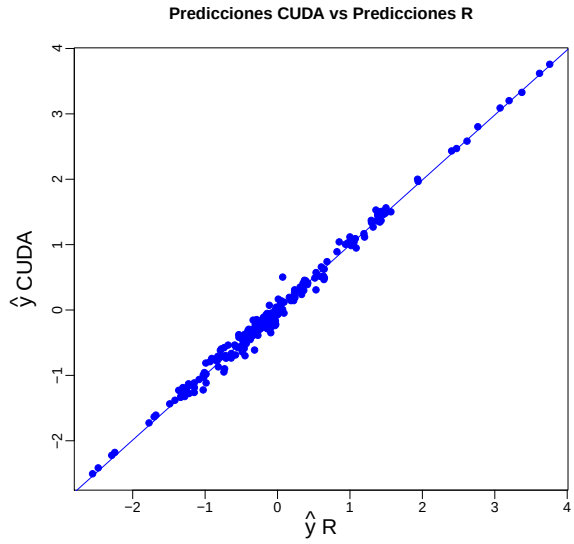

Figura 5.5: Relación de los valores predichos entre el software desarrollado y la biblioteca brnn del paquete estadístico R.

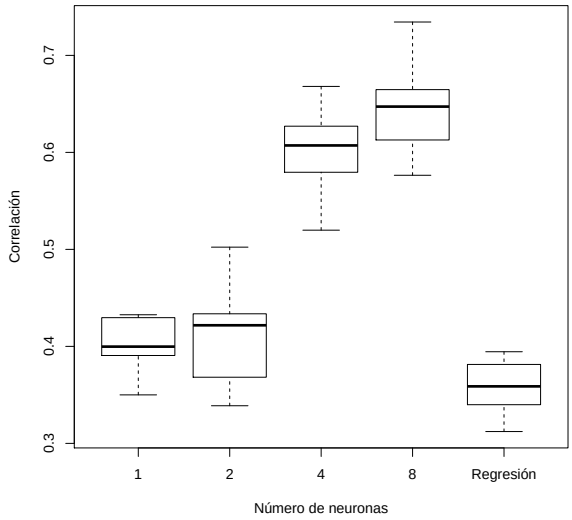

Figura 5.6: Diagrama de cajas de las correlaciones Pearson del entrenamiento.

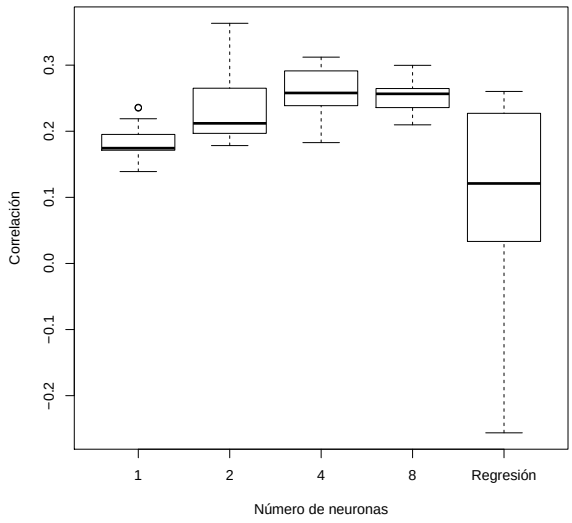

Figura 5.7: Diagrama de cajas de las correlaciones Pearson de la prueba.

Cuadro 5.3: Correlación Pearson entre los valores predichos y los observados (entrenamiento y prueba).

	Conjunto de datos "Million Song Dataset" (1000 observaciones)									
	1 Neurona		2 Neuronas		4 Neuronas		8 Neuronas		Regresión Lineal	
Conjunto	Entrenamiento	Prueba	Entrenamiento	Prueba	Entrenamiento	Prueba	Entrenamiento	Prueba	Entrenamiento	Prueba
1	0.3715	0.177	0.5023	0.2097	0.5198	0.226	0.5764	0.257	0.3334789	0.2274125
2	0.4298	0.1717	0.3841	0.2445	0.595	0.216	0.6523	0.2526	0.3398762	-0.002499683
3	0.4153	0.1524	0.3637	0.2281	0.6149	0.2499	0.6902	0.264	0.3641777	0.05990433
4	0.3908	0.1715	0.4288	0.2703	0.5795	0.2268	0.6621	0.2347	0.3657606	0.1742224
5	0.4296	0.1789	0.3588	0.2004	0.588	0.2773	0.5905	0.2593	0.3944803	0.2271206
6	0.4301	0.1717	0.4536	0.3234	0.5799	0.2417	0.6208	0.2836	0.3813831	0.03331754
7	0.3907	0.1713	0.3415	0.1841	0.6262	0.2588	0.6473	0.2644	0.3888116	0.2602058
8	0.3906	0.188	0.3427	0.1848	0.6173	0.2612	0.5853	0.2645	0.3122002	0.1150877
9	0.3994	0.2183	0.43	0.1946	0.5758	0.3091	0.6575	0.2141	0.3464094	-0.256224
10	0.3597	0.2353	0.3709	0.1968	0.6258	0.2387	0.6813	0.2604	0.3533668	0.126911

CAPÍTULO 6. CONCLUSIONES Y RECOMENDACIONES

La presente tesis propone el uso de GPUs para ajustar Redes Neuronales Artificiales Regularizadas Bayesianas y evaluar su desempeño. En función de los resultados obtenidos se puede concluir que el desempeño fue aceptable ya que las predicciones mostraron una correlación similar a la obtenida con las otras implementaciones. Respecto a los tiempos de cómputo, se concluye que a pesar del ahorro en tiempo de desarrollo que se puede conseguir con la utilización de bibliotecas especializadas, es necesario realizar un proceso más exhaustivo con el fin de disminuir los tiempos de traslado de datos entre dispositivos, si bien se consigue esto en la versión 2 de la aplicación que utiliza CUDA, se debe hacer un uso más optimizado de los intercambios y el espacio de memoria, sobre todo porque los datos, a medida que incrementa su tamaño, requieren una mayor cantidad de esta y no es posible contener todas las matrices resultantes en la GPU.

El cómputo paralelo, particularmente el que se realiza con la ayuda de GPUs se ha vuelto una herramienta que permite trabajar con conjuntos de datos que hace algunos años o en arquitecturas más sencillas sería muy complejo o hasta prohibitivo realizarlos. La realización de la aplicación presentada en este proyecto muestra que se es posible realizar investigaciones sin recurrir a centros de cómputo de alto rendimiento y los resultados pueden ser replicables y escalables.

6.1 Temas de investigación futuros

La arquitectura CUDA posee un gran número de características que requieren más tiempo de estudio del que se contaba para esta investigación, se sugiere explorar de manera exhaustiva el manejo de memoria ya que, este es uno de los puntos claves en el diseño de aplicaciones para este tipo de dispositivos esto, porque a diferencia de las aplicaciones que sólo hacen uso de la unidad central de procesamiento (CPU), implica una planificación distinta de los procedimientos para reducir el número de traslados de datos entre dispositivos. Se sugiere también explorar el desarrollo de aplicaciones híbridas, las cuales combinan el uso de CPU y GPU y suelen ser la mejor alternativa para aprovechar todos los recursos de un sistema heterogéneo.

CAPÍTULO 7. REFERENCIAS

Aggarwal, C. C. (2020). *Linear Algebra and Optimization for Machine Learning. A Textbook*. Springer, USA.

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A. y Sorensen, D. (1987). *LAPACK Users'Guide*. Society for Industrial and Applied Mathematics, USA, tercera edición.

Benner, P., Ezzatti, P., Quintana-Ortí, E. y Remón, A. (2013). Matrix inversion on CPU-GPU platforms with applications in control theory. *Concurrency and Computation: Practice and Experience*, 25, 8, 1170–1182.

Berk, R. (2011). *Statistical Learning from a Regression Perspective, Second Edition.*. Springer, USA.

Bertin-Mahieux, T., Ellis, D. P., Whitman, B. y Lamere, P. (2011). The million song dataset. En *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*.

Bryant, R. E. y O'Hallaron, D. R. (2016). *Computer Systems: A Programmer's Perspective*. Pearson Education, USA, tercera edición.

Cheng, J., Grossman, M. y McKercher, T. (2014). *Professional CUDA C programing.*. Wrox, USA.

Demuth, H. B., Beale, M. H., De Jess, O. y Hagan, M. T. (2014). *Neural Network Design*. Martin Hagan, Stillwater, OK, USA, segunda edición. ISBN 0971732116.

Du, K. y Swamy, M. N. S. (2014). *Neural Networks and Statistical Learning*. Springer,, USA.

Dua, D. y Graff, C. (2017). UCI machine learning repository.

Elahi, A. (2018). *Computer Systems. Digital Design, Fundamentals of Computer Architecture and Assembly Language*. Springer, Cham, USA.

Farber, R. (2011). *CUDA Application Design, and Development*. Morgan Kaufmann, USA.

Fausett, L. V. (1993). *Fundamentals of Neural Networks: Architectures, Algorithms and Applications.*. Pearson, USA.

Fayez, G. (2011). *Algorithms and Parallel Computing*. Wiley, USA.

Foresse, F. D. y Hagan, M. T. (1997). Gauss-Newton Approximation to Bayesian learning. *Hagan.ecen.ceat.okstate.edu*.

Freeman, J. A. y Skapura, D. M. (1991). *Neural Networks Algorithms, Applications, and Programming Techniques*. Addison-Wesley, USA.

Fyfe, C. (1996). *Artificial Neural Networks*. The University of Paisley, USA.

Gentle, J. E. (2017). *Matrix Algebra. Theory, Computations and Applications in Statistics*. Springer, USA, segunda edición.

Gianola, D., Hayrettin, O., Weigel, K. y Rosa, G. (2011). Predicting complex cuantitative traits with Bayesian neural networks: a case study with Jersey cows and wheat. *BMC Genetics*, 12, 87.

Golub, G. H. y Van Loan, C. F. (2013). *Matrix Computations*. The Johns Hopkins University Press, USA, cuarta edición.

Guzmán, E., Vázquez, M., del Valle, D. y Pérez-Rodríguez, P. (2018). Artificial Neuronal Networks: A Bayesian Aproach Using Parallel Computing. *Revista Colombiana de estadística*, 41, 2, 173–189.

Haidar, A., Abdelfatah, A., Tomov, S. y Dongarra, J. (2016). High-performance cholesky factorization for gpu-only execution. En *GPGPU-10: Proceedings of the General Purpose GPUs*, 42–52. Association for Computing Machinery, New York, NY, USA.

Heaton, J. (2008). *Introduction to Neural Networks with C#*. Heaton Research, Inc, USA, segunda edición.

Hennessy, J. L. y Patterson, D. A. (2019). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, USA, 6ª edición.

Heumann, C., Schomaker, M. y Shalabh (2016). *Introduction to Statistics and Data Analysis*. Springer, USA.

Hollander, M. y Wolfe, D. A. (1999). *Nonparametric Statistical Methods*. Wiley, segunda edición.

Isasi, P. y Galván, I. M. (2004). *Redes de neuronas artificiales, un enfoque práctico*. Pearson. Madrid, España.

James, G., Witten, D., Hastie, T. y Tibshirani, R. (2013). *An introduction to statistical learning with aplications in R*. Springer, USA.

Kayri, M. (2016). Predictive Abilities of Bayesian Regularization and Levenberg-Marquardt Algorithms in Artificial Neural Networks: A Comparative Empirical Study on Social Data. *Mathematical and Computational Applications*, 21, 20(2).

Kirk, D. B. y Hwu, W. W. (2017). *Programming Massively Parallel Processors. A Hands-on Approach*. Morgan Kaufman, USA, tercera edición.

Kolmogorov, A. K. (1957). On the Representation of Continuous Functions of Several Variables by Superposition of Continuous Functions of One Variable and Addition. *Doklady Akademii Nauk SSSR*, 114, 369–373.

Li, P., Xu, M., Wu, J. y Shang, L. (2016). Using canonical correlation analysis for parallelized attribute reduction. En R. Booth y M.-L. Zhang, eds., *PRICAI 2016: Trends in Artificial Intelligence*, 433–445. Springer International Publishing, Cham. ISBN 978-3-319-42911-3.

Li, X., Schissler, A. G., Wu, R., Barford, L. y Harris, F. C. (2019). A graphical processing unit accelerated normal to anything algorithm for high dimensional multivariate simulation. En S. Latifi, ed., *16th International Conference on Information Technology-New Generations (ITNG 2019)*, 339–345. Springer International Publishing, Cham. ISBN 978-3-030-14070-0.

Lukarski, D. y Neytcheva, M. (2014). On the impact of the heterogeneous multicore and many-core platforms on iterative solution methods and preconditioning techniques. En E. Jeannot y J. Ẑilinskas, eds., *High-Performance Computing on Complex Environments*. Wiley, USA.

MacKay, D. J. C. (1992a). Bayesian Interpolation. *Neural Computation*, 4, 415–447.

McClelland, J. y Rumelhart, D. (1986). *Explorations in Parallel Distributed Processing*. The MIT Press, USA.

Mehrotra, K., Mohan, C. K. y Ranka, S. (1996). *Elements of Artificial Neural Networks*. The MIT Press, USA.

Meyer, K. y Tier, B. (2013). Utility Of Graphics Processing Units For Dense Matrix Calculations In Computing And Inverting Genomic Relationship Matrices. *Assoc. Advmt. Anim. Breed. Genet. Napier, New Zealand*, 20, 270–273.

Minsky, M. y Papert, S. (1969). *Perceptrons: An introduction to computational geometry*. The MIT Press, USA.

Neri, F. (2019). *Linear Algebra for Computational Sciences and Engineering*. Springer, Switzerland, segunda edición.

Okut, H. (2016). Bayesian regularized neural networks for small n big p data. En J. Rosa, ed., *Artificial Neural Networks-Models and Applications*. IN-TECH, Munich, Germany.

Paciorek, C. J. y Schervish, M. J. (2003). Nonstationary covariance functions for gaussian process regression. En *Proceedings of the 16th International Conference on Neural Information Processing Systems*, NIPS'03, 273–280. MIT Press, Cambridge, MA, USA.

Parker, D. (1985). Learning logic. Tr-47, Center for Computational Research in Economics and Management Science, Massachusetts Institute of Technology, Cambridge, MA.

Patterson, D. A. y Hennessy, J. L. (2014). *Computer Organization and Design. The hardware / software interface*. Morgan Kauffman, USA.

Pérez-Rodríguez, P. y Gianola, D. (2020). *brnn: Bayesian Regularization for Feed-Forward Neural Networks*. R package version 0.8.

Pérez-Rodríguez, P., Gianola, D., Weigel, K. A., Rosa, G. J. M. y Crossa, J. (2013). Technical Note: An R package for fitting Bayesian regularized neural networks with applications in animal breeding. *Journal of animal science.*, 91, 3522:3525.

Ploskas, N. y Samaras, N. (2016). *GPU Programming in MATLAB*. Morgan Kaufmann, USA.

R Core Team (2020). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Sanders, J. y Kandrot, E. (2011). *CUDA by example : an introduction to general-purpose GPU programming*. Addison-Wesley, USA.

Schindler, A. y Rauber, A. (2014). Capturing the temporal domain in echonest features for improved classification effectiveness. En A. Nürnberger, S. Stober, B. Larsen y M. Detyniecki, eds., *Adaptive Multimedia Retrieval: Semantics, Context, and Adaptation*, 214–227. Springer International Publishing, Cham. ISBN 978-3-319-12093-5.

Sengupta, S., Harris, M., Zhang, Y. y Owens, J. D. (2007). *Scan Primitives for GPU Computing*. Graphics Hardware.

Shalev-Shwartz, S. y Ben-David, S. (2014). *Understanding Machine Learning, From Theory to Algorithms*. Cambridge University Press, USA.

Soyata, T. (2018). *GPU Parallel Program Development Using CUDA*. Computational Science Series. CRC, USA.

Stallings, W. (2016). *Computer Organization and Architecture Designing for Performance*. Pearson Education, USA, 10ª edición.

Storti, D. y Yurtoglu, M. (2016). *CUDA for Engineers, An Introduction to High-Performance Parallel Computing*. Addison-Wesley, USA.

Sun, Z., Chen, Y., Lin, X., Qin, X. y Wang, H. (2017). A Bayesian regularized artificial neural network for adaptive optics forecasting. *Optics Communications*, 382, 519–527.

Suzuki, K. (2013). *Artificial Neural Networks - Architectures and Applications*. InTech, Croacia.

Vaidya, B. (2018). *Hands-On GPU-Accelerated Computer Vision with OpenCV and CUDA,Effective techniques for processing complex image data in real time using GPUs*. Packt Publishing, United Kingdome.

Werbos, P. y John, P. (1974). Beyond regression : new tools for prediction and analysis in the behavioral sciences. *Thesis (Ph. D.)*.

Widrow, B. y Hoff, M. E. (1960). Adaptive switching circuits. En *IRE WESCON Convention Record*, 96–104.

Wilt, N. (2013). *The CUDA Handbook, A Comprehensive Guide to GPU Programming*. Addison Wesley, USA.

Zhang, X.-D. (2020). *A Matrix Algebra Approach to Artificial Intelligence*. Springer, Singapore.

ANEXOS

Anexo A: Código en *C/CUDA* con las bibliotecas *cuBLAS* y *cuSolver*

El siguiente código en el lenguaje de programación *C/CUDA* contiene la implementación del programa para la regularización Bayesiana de una red neuronal artificial. El código hace uso de las bibliotecas para álgebra lineal *cuBLAS* y *cuSolver*.

```
#include <stdio.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#include <cusolverDn.h>


void print_array_column_major(double *in, int nrows, int ncolumns)
{
  int i,j;

  for(i=0; i<nrows;i++)
  {
    for(j=0; j<ncolumns;j++)
    {
      printf("%f \t",in[j*nrows+i]);
    }
    printf("\n");
  }
}

void welcome()
{
  printf("\n");
  printf("#_____________________________________#\n");
  printf("# trainbr-CUDA                       #\n");
  printf("# Bayesian regularized neural networks   #\n");
  printf("#_____________________________________#\n");
  printf("\n");
}

double cor_Pearson(double *x, double *y, int n)
{
   double sumx=0;
```

```
34    double sumy=0;
35    double sumxy=0;
36    double sumx2=0;
37    double sumy2=0;
38    int i;
39
40    for(i=0; i<n;i++)
41    {
42       sumx+=x[i];
43       sumy+=y[i];
44       sumxy+=x[i]*y[i];
45       sumx2+=x[i]*x[i];
46       sumy2+=y[i]*y[i];
47    }
48    return((sumxy-sumx*sumy/(double (n)))/(pow((sumx2-sumx*sumx/(double (n)))*(sumy2-
       sumy*sumy/(double (n))),0.5)));
49 }
50
51 void read_matrix(double *X, int rows, int columns, char *Input_file)
52 {
53     const int MAX_LEN = 32767;
54     char Buffer[MAX_LEN];
55     char *token=NULL;
56     FILE *ptr;
57     int i,j;
58
59     ptr=fopen(Input_file,"r");
60     if(ptr!=NULL)
61     {
62       i=0;
63   printf("Loading␣incidence␣matrix...");
64         while(fgets(Buffer, MAX_LEN, ptr) != NULL)
65       {
66           token=strtok(Buffer,",");
67           j=0;
68           while(token!=NULL)
69     {
70             X[i+(j*rows)]=atof(token);
71             token = strtok(NULL,",");
72             j++;
73          }
74          i++;
75       }
76       printf("Done\n");
77         if(i!=rows)
78         {
79           printf("The␣file␣has␣MORE/LESS␣lines␣that␣those␣indicated␣by␣the␣user\n");
80           exit(1);
81         }
82         fclose(ptr);
83     }else{
84         printf("Unable␣to␣open␣input␣file␣with␣incidence␣matrix\n");
85   exit(1);
86     }
```

```
87 }
88
89 void read_vector(double *y, int rows,char *Input_file)
90 {
91       const int MAX_LEN=256;
92       char Buffer[MAX_LEN];
93       int i;
94       FILE *ptr;
95       ptr=fopen(Input_file,"r");
96       if(ptr!=NULL)
97       {
98   i=0;
99   printf("Loading␣response␣vector...");
100  while(fgets(Buffer, MAX_LEN, ptr) != NULL)
101  {
102      y[i]=atof(Buffer);
103      i++;
104      }
105      printf("Done\n");
106  if(i!=rows)
107        {
108      printf("The␣file␣has␣MORE/LESS␣lines␣that␣those␣indicated␣by␣the␣user\n");
109            exit(1);
110        }
111        fclose(ptr);
112      }else{
113    printf("Unable␣to␣open␣file␣with␣response␣vector\n");
114    exit(1);
115      }
116 }
117
118 double tansig(double x)
119 {
120     return(2.0/(1.0+exp(-2.0*x)) - 1.0);
121 }
122
123 double sech(double x)
124 {
125     return(2.0*exp(x)/(exp(2.0*x)+1.0));
126 }
127
128 double predictions_nn(double *X, unsigned int rows,
129                       unsigned int columns, double *theta,
130                       unsigned int neurons,double *yhat, double *y,
131                       double *residuals)
132 {
133    unsigned int i,j,k;
134    double suma,z;
135    double error=0;
136
137    for(i=0;i<rows;i++)
138    {
139       suma=0;
140       for(k=0;k<neurons;k++)
```

```
141       {
142    z=0;
143    for(j=0;j<columns;j++)
144    {
145       z+=X[i+(j*rows)]*theta[(columns+2)*k+j+2];
146    }
147    z+=theta[(columns+2)*k+1];
148    suma+=theta[(columns+2)*k]*tansig(z);
149       }
150       yhat[i]=suma;
151       residuals[i]=y[i]-yhat[i];
152       error+=pow(residuals[i],2.0);
153    }
154    return(error);
155 }
156
157 void predictions_nn(double *X, unsigned int rows,
158                     unsigned int columns, double *theta,
159                     unsigned int neurons,double *yhat)
160 {
161    unsigned int i,j,k;
162    double suma,z;
163
164    for(i=0;i<rows;i++)
165    {
166       suma=0;
167       for(k=0;k<neurons;k++)
168       {
169    z=0;
170    for(j=0;j<columns;j++)
171    {
172       z+=X[i+(j*rows)]*theta[(columns+2)*k+j+2];
173    }
174    z+=theta[(columns+2)*k+1];
175    suma+=theta[(columns+2)*k]*tansig(z);
176       }
177       yhat[i]=suma;
178    }
179 }
180
181 void jacobian(double *X, unsigned int n,
182               unsigned int p,double *theta,
183               unsigned int neurons,double *J)
184 {
185    unsigned int i,j,k;
186    double z,dtansig;
187
188    printf("Calulating␣jacobian...");
189    for(i=0; i<n; i++)
190    {
191      for(k=0; k<neurons; k++)
192      {
193     z=0;
194     for(j=0;j<p;j++)
```

```
195     {
196         z+=X[i+(j*n)]*theta[(p+2)*k+j+2];
197     }
198     z+=theta[(p+2)*k+1];
199     dtansig=pow(sech(z),2.0);
200     J[i+(((p+2)*k)*n)]=-tansig(z);
201     J[i+(((p+2)*k+1)*n)]=-theta[(p+2)*k]*dtansig;
202 
203    for(j=0; j<p;j++)
204    {
205        J[i+(((p+2)*k+j+2)*n)]=-theta[(p+2)*k]*dtansig*X[i+(j*n)];
206    }
207      }
208    }
209    printf("Done\n");
210 }
211 
212 double sc(double *theta, int npar)
213 {
214    double suma=0;
215    for(int i=0; i<npar; i++) suma+=pow(theta[i],2.0);
216    return(suma);
217 }
218 
219 void initnw(double *theta, int n, int p, int neurons, int npar)
220 {
221   double scaling_factor,dx, suma;
222   int k,j;
223 
224   srand(time(NULL));
225   for(int i=0; i<npar;i++)
226   theta[i]=((double) rand()) / ((double) RAND_MAX)-0.5;
227 
228   printf("Initializing␣using␣the␣Nguyen-Widrow␣method\n");
229   if(p==1)
230   {
231       scaling_factor=0.7*neurons;
232       for(k=0;k<neurons;k++)
233       {
234     theta[(p+2)*k+1]=scaling_factor*(1.0-2.0*(((double) rand())
235          / ((double) RAND_MAX)));
236     theta[(p+2)*k+2]=scaling_factor;
237       }
238   }else{
239      scaling_factor=0.7*pow(neurons,(1.0/double (n)));
240      dx=2.0/double(neurons);
241      for(k=0; k<neurons;k++)
242      {
243           suma=0;
244     for(j=0;j<p;j++)
245     {
246       suma+=pow(theta[(p+2)*k+2+j],2.0);
247     }
248     for(j=0; j<p; j++)
```

```
249     {
250       theta[(p+2)*k+2+j]=scaling_factor*theta[(p+2)*k+2+j]/pow(suma,0.5);
251     }
252     theta[(p+2)*k+1]=(-1.0+dx*k)*scaling_factor;
253      }
254   }
255 }
256
257 /*
258  * Crossprod of a matrix using cublas
259  * C=A'A
260  */
261
262 void matrix_crossprod(double *A, int row_a, int col_a, double *C, int row_c, int col_c
      )
263 {
264     double alpha=1.0;
265     double beta=0.0;
266
267     /*cuda malloc status*/
268     cudaError_t cudaStat;
269
270     /*cublas functions status*/
271     cublasStatus_t stat;
272
273     /*cublas handle status*/
274     cublasHandle_t handle;
275
276     /*Memory allocation in the device*/
277     double *d_A;
278     double *d_C;
279
280     cudaStat=cudaMalloc((void**)&d_A,row_a*col_a*sizeof(double)); //device memory
       allocation for d_A
281     cudaStat=cudaMalloc((void**)&d_C,row_c*col_c*sizeof(double)); //device memory
       allocation for d_C
282
283     stat = cublasCreate(&handle); // initialize CUBLAS context
284
285     cudaMemcpy(d_A,A,row_a*col_a*sizeof(double),cudaMemcpyHostToDevice);
286     cudaMemcpy(d_C,C,row_c*col_c*sizeof(double),cudaMemcpyHostToDevice);
287
288      stat=cublasDgemm(handle,
289                       CUBLAS_OP_T,
290                       CUBLAS_OP_N,
291                       col_a,
292                       col_a,
293                       row_a,
294                       &alpha,
295                       d_A,
296                       row_a,
297                       d_A,
298                       row_a,
299                       &beta,
```

```
300                         d_C,
301                         col_a
302                         );
303 
304     if(stat!=CUBLAS_STATUS_SUCCESS)
305     {
306   printf("Error in cublas\n");
307   exit(1);
308     }
309 
310     cudaMemcpy(C,d_C,row_c*col_c*sizeof(double),cudaMemcpyDeviceToHost);
311 
312     /*Free memory*/
313     cudaFree(d_A);
314     cudaFree(d_C);
315     cublasDestroy(handle);
316 }
317 
318 /*
319  * Matrix vector product using cublas
320  * sol=A'b
321  */
322 void matrix_vector_product(double *A, int row_a, int col_a, double *b,  double *sol)
323 {
324     int one=1;
325     double alpha=1.0;
326     double beta=0.0;
327 
328     /*cuda malloc status*/
329     cudaError_t cudaStat;
330 
331     /*cublas functions status*/
332     cublasStatus_t stat;
333 
334     /*cublas handle status*/
335     cublasHandle_t handle;
336 
337     /*Memory allocation in the device*/
338     double *d_A;
339     double *d_b;
340     double *d_sol;
341 
342     cudaStat=cudaMalloc((void**)&d_A,row_a*col_a*sizeof(double)); //device memory
      allocation for d_A
343     cudaStat=cudaMalloc((void**)&d_b,row_a*sizeof(double));       //device memory
      allocation for d_b
344     cudaStat=cudaMalloc((void**)&d_sol,col_a*sizeof(double));   //device memory
      allocation for d_sol
345 
346     stat = cublasCreate(&handle); // initialize CUBLAS context
347 
348     cudaMemcpy(d_A,A,row_a*col_a*sizeof(double),cudaMemcpyHostToDevice);
349     cudaMemcpy(d_b,b,row_a*sizeof(double),cudaMemcpyHostToDevice);
350 
```

```
351     stat=cublasDgemv(handle,
352                      CUBLAS_OP_T,
353                      row_a,
354                      col_a,
355                      &alpha,
356                      d_A,
357                      row_a,
358                      d_b,
359                      one,
360                      &beta,
361                      d_sol,
362                      one);
363 
364     cudaMemcpy(sol,d_sol,col_a*sizeof(double),cudaMemcpyDeviceToHost);
365 
366     /*Free memory*/
367     cudaFree(d_A);
368     cudaFree(d_b);
369     cudaFree(d_sol);
370     cublasDestroy(handle);
371 }
372 
373 /*
374 Solve a linear system of equations using cuda Ax=b,
375 The solution x, will be given in vector b, so the original one will be lost
376 */
377 
378 void solve_Cholesky(double *A, int row_a, int col_a, double *b)
379 {
380    cudaError cudaStatus;
381    cusolverStatus_t cusolverStatus;
382    cusolverDnHandle_t handle;
383 
384    double *d_A, *d_b, *Work; // matrix A, rhs b and worksp.
385    int *d_info, Lwork; // device version of info, worksp.size
386    int info_gpu = 0; // device info copied to host
387 
388    cudaStatus = cudaGetDevice(0);
389    cusolverStatus = cusolverDnCreate(&handle); // create handle
390    cublasFillMode_t uplo = CUBLAS_FILL_MODE_LOWER;
391 
392    // prepare memory on the device
393    cudaStatus = cudaMalloc((void**)&d_A, row_a*row_a*sizeof(double));
394    cudaStatus = cudaMalloc((void**)&d_b, row_a*sizeof(double));
395    cudaStatus = cudaMalloc((void**)&d_info, sizeof(int));
396    cudaStatus = cudaMemcpy(d_A, A, row_a*row_a*sizeof(double),cudaMemcpyHostToDevice);
       // copy A->d_A
397    cudaStatus = cudaMemcpy(d_b, b, row_a*sizeof(double),cudaMemcpyHostToDevice); //
       copy b->d_b
398 
399    // compute workspace size and prepare workspace
400    cusolverStatus = cusolverDnDpotrf_bufferSize(handle, uplo, row_a, d_A,row_a, &Lwork
       );
401 
```

```
402     cudaStatus = cudaMalloc((void**)&Work,Lwork*sizeof(double));
403
404     // Cholesky decomposition d_A=L*L^T, lower triangle of d_A is
405     // replaced by the factor L
406     cusolverStatus = cusolverDnDpotrf(handle,uplo,row_a,d_A,row_a,Work, Lwork,d_info);
407
408     if(cusolverStatus==CUSOLVER_STATUS_SUCCESS)
409     {
410       printf("Successful␣Cholesky␣factorization\n");
411     }
412
413     // solve d_A*x=d_b, where d_A is factorized by potrf function
414     // d_b is overwritten by the solution
415     cusolverStatus = cusolverDnDpotrs(handle,uplo,row_a, 1,d_A,row_a, d_b,row_a,d_info)
        ;
416     if(cusolverStatus==CUSOLVER_STATUS_SUCCESS)
417     {
418        printf("Successful␣solution\n");
419     }
420
421     cudaStatus = cudaDeviceSynchronize();
422
423     cudaStatus = cudaMemcpy(b, d_b, row_a*sizeof(double),cudaMemcpyDeviceToHost ); //
        copy solution to host d_b ->b
424
425     // free memory
426     cudaFree(d_A);
427     cudaFree(d_b);
428     cudaFree(d_info);
429     cudaFree(Work);
430     cusolverStatus = cusolverDnDestroy(handle);
431     cudaStatus = cudaDeviceReset();
432
433 }
434
435 /*
436  * Invert a matrix using cuda
437  * WARNING: Only the UPPER part is useful
438  */
439
440 void inverse(double *A, int row_a, double *Ainv)
441 {
442
443     cudaError cudaStatus;
444     cusolverStatus_t cusolverStatus;
445     cusolverDnHandle_t handle;
446
447     double *d_A, *d_Ainv, *Work; // matrix A and worksp.
448     int *d_info, Lwork; // device version of info, worksp.size
449     int info_gpu = 0; // device info copied to host
450
451     cudaStatus = cudaGetDevice(0);
452     cusolverStatus = cusolverDnCreate(&handle); // create handle
453     cublasFillMode_t uplo = CUBLAS_FILL_MODE_LOWER;
```

```
454
455     // prepare memory on the device
456     cudaStatus = cudaMalloc((void**)&d_A, row_a*row_a*sizeof(double));
457     cudaStatus = cudaMalloc((void**)&d_Ainv,row_a*row_a*sizeof(double));
458     cudaStatus = cudaMalloc((void**)&d_info, sizeof(int));
459     cudaStatus = cudaMemcpy(d_A, A, row_a*row_a*sizeof(double),cudaMemcpyHostToDevice);
          // copy A->d_A
460     cudaStatus = cudaMemcpy(d_Ainv,Ainv,row_a*row_a*sizeof(double),
         cudaMemcpyHostToDevice); // copy Ainv->d_Ainv
461
462     // compute workspace size and prepare workspace
463     cusolverStatus = cusolverDnDpotrf_bufferSize(handle, uplo, row_a, d_A,row_a, &Lwork
         );
464
465     cudaStatus = cudaMalloc((void**)&Work,Lwork*sizeof(double));
466
467     // Cholesky decomposition d_A=L*L^T, lower triangle of d_A is
468     // replaced by the factor L
469     cusolverStatus = cusolverDnDpotrf(handle,uplo,row_a,d_A,row_a,Work, Lwork,d_info);
470
471     // solve d_A*x=d_b, where d_A is factorized by potrf function
472     // d_Ainv is overwritten by the solution
473      cusolverStatus = cusolverDnDpotrs(handle,uplo,row_a, row_a,d_A,row_a, d_Ainv,row_a
         ,d_info);
474
475     cudaStatus = cudaDeviceSynchronize();
476
477     cudaStatus = cudaMemcpy(Ainv, d_Ainv, row_a*row_a*sizeof(double),
         cudaMemcpyDeviceToHost ); // copy solution to host d_Ainv -> Ainv
478
479     // free memory
480     cudaFree(d_A);
481     cudaFree(d_Ainv);
482     cudaFree(d_info);
483     cudaFree(Work);
484     cusolverStatus = cusolverDnDestroy(handle);
485     cudaStatus = cudaDeviceReset();
486
487 }
488
489
490 int main(int argc, char **argv)
491 {
492   int i,j;
493       double *X, *y,*yhat, *residuals,*theta, *theta_new,
494                 *J, *H, *delta, *Hinv, *g, *diag_H, *F_history;
495
496   double gamma;
497       double Ed, Ed_new;
498       double Ew, Ew_new;
499       double alpha;
500       double beta;
501     double C;
502       double C_new;
```

```
503       int epoch=0;
504       int max_epochs=1000;
505       double mu=0.005;
506       double mu_inc=10.0;
507       double mu_dec=0.1;
508       double mu_max=1e10;
509       double change=0.001;
510       double trace;
511       double t1, t2, t3, t4;
512
513   /*
514   Read command line arguments, pass 1
515   */
516   int n=atoi(argv[1]);
517       int p=atoi(argv[2]);
518       int neurons=atoi(argv[3]);
519   int npar=neurons*(2+p);
520
521   /*
522   End of reading command line arguments, pass 1
523   */
524
525   /*
526   Initialization
527   */
528
529   welcome();
530
531   printf("Cases:␣%d\n",n);
532         printf("Predictors:␣%d\n",p);
533   printf("Neurons:␣%d\n",neurons);
534   printf("Number␣of␣parameters␣to␣estimate:␣%d\n",npar);
535
536   X = (double *) malloc(n*p*sizeof(double));
537   y = (double *) malloc(n*sizeof(double));
538   yhat=(double *) malloc(n*sizeof(double));
539   residuals=(double *) malloc(n*sizeof(double));
540   J=(double *) malloc(n*npar*sizeof(double));
541   H=(double *) malloc(npar*npar*sizeof(double));
542   Hinv=(double *) malloc(npar*npar*sizeof(double));
543   theta=(double *) malloc(npar*sizeof(double));
544   theta_new=(double *) malloc(npar*sizeof(double));
545   delta=(double *) malloc(npar*sizeof(double));
546   g=(double *) malloc(npar*sizeof(double));
547   diag_H=(double *) malloc(npar*sizeof(double));
548   F_history=(double *) malloc(max_epochs*sizeof(double));
549
550   read_matrix(X,n,p,argv[4]);
551   read_vector(y,n,argv[5]);
552
553   initnw(theta,n,p,neurons,npar);
554
555   gamma=npar;
556         Ed=predictions_nn(X,n,p,theta,neurons,yhat,y,residuals);
```

```
557         beta=(n-gamma)/(2.0*Ed);
558         if(beta<0) beta=1.0;
559         Ew=sc(theta,npar);
560         alpha=gamma/(2.0*Ew);
561 
562 /*
563 End of initialization
564 */
565 
566 /*
567 main loop
568 */
569 
570     int flag_C;
571     int flag_mu=1;
572     int flag_change_F=1;
573 
574     while(epoch<max_epochs && flag_mu && flag_change_F)
575 {
576      printf("epoch=%d\n",epoch);
577      printf("alpha=%4.4f\tbeta=%4.4f\n",alpha,beta);
578 
579      /*Calculate the Jacobian*/
580            jacobian(X,n,p,theta,neurons,J);
581 
582            //print_array_column_major(J,n,npar);
583            //exit(1);
584 
585      /*Calculate the residuals*/
586      Ed=predictions_nn(X, n, p, theta,neurons,yhat,y,residuals);
587      Ew=sc(theta,npar);
588      printf("Ed=%4.4f\tEw=%4.4f\n",Ed,Ew);
589 
590      /*
591            Calculate the Hessian
592            */
593            memset(Hinv,0,npar*npar*sizeof(double));
594            matrix_crossprod(J,n,npar,H, npar, npar);
595 
596      //print_array_column_major(H,npar,npar);
597            //exit(1);
598 
599      /*
600             Calculate the gradient
601             Extract the diagonal of H
602             Compute the cost
603            */
604 
605            matrix_vector_product(J,n,npar,residuals,g);
606 
607            //print_array_column_major(g,npar,1);
608            //exit(1);
609 
610      for(i=0; i<npar;i++)
```

```
611         {
612            g[i]=g[i]+alpha/beta*theta[i];
613            //printf("%f\n",g[i]);
614            diag_H[i]=H[i+(i*npar)];
615         }
616 
617         C=beta*Ed+alpha*Ew;
618 
619               flag_C=1;
620 
621         while(flag_C && flag_mu)
622               {
623     for(i=0; i<npar;i++)
624       {
625            H[i+(i*npar)]=diag_H[i]+alpha/beta+mu/(2.0*beta);
626            delta[i]=g[i];
627           }
628 
629     //print_array_column_major(delta,npar,1);
630                 //exit(1);
631 
632         solve_Cholesky(H,npar,npar,delta);
633     //print_array_column_major(delta,npar,1);
634                 //exit(1);
635 
636         for(i=0; i<npar;i++)
637         {
638             theta_new[i]=theta[i]-delta[i];
639         }
640 
641         Ed_new=predictions_nn(X, n, p, theta_new,neurons,yhat,y,residuals);
642         Ew_new=sc(theta_new,npar);
643         C_new=beta*Ed_new+alpha*Ew_new;
644         printf("C_new=%f\tmu=%f\n",C_new,mu);
645 
646         if(C_new<C)
647         {
648     printf("C_new␣smaller␣than␣C\n");
649           mu=mu*mu_dec;
650           if (mu < 1e-20) mu = 1e-20;
651           flag_C=0;
652           F_history[epoch]=C_new;
653           if(epoch>3)
654           {
655             if((fabs(F_history[epoch]-F_history[epoch-1])<change) && (fabs(F_history[
    epoch-1]-F_history[epoch-2])<change)  && (fabs(F_history[epoch-2]-F_history[epoch
    -3])<change))
656             {
657          flag_change_F=0;
658          printf("Changes␣in␣F=␣beta*SCE␣+␣alpha*Ew␣in␣last␣3␣iterations␣less␣than␣
    0.001\n");
659             }
660          }
661         }else{
```

```
662                 printf("C_new bigger than C\n");
663             mu=mu*mu_inc;
664             if(mu>mu_max)
665             {
666               flag_mu=0;
667               printf("Maximum mu reached\n");
668             }
669           }
670         } //End of while
671 
672               /*
673                * Update all
674               */
675 
676               //printf("Updating all...\n");
677         for(i=0; i<npar;i++)
678         {
679            for(j=0; j<npar;j++)
680            {
681              if(i==j){
682           H[i+(i*npar)]=2.0*beta*diag_H[i]+2.0*alpha;
683               }else{
684           H[i+(j*npar)]=2.0*beta*H[i+(j*npar)];
685               }
686            }
687          }
688 
689          //set Hinv to  identity
690                memset(Hinv,0,npar*npar*sizeof(double));
691                for(i=0;i<npar;i++)
692                {
693                    Hinv[i+(i*npar)]=1.0;
694                }
695 
696                //Hinv is overwitten with the inverse...
697                inverse(H,npar,Hinv);
698 
699          trace=0;
700          for(i=0; i<npar; i++)
701          {
702             theta[i]=theta_new[i];
703             trace+=Hinv[i+(i*npar)];
704          }
705 
706 
707          gamma=npar-2.0*alpha*trace;
708          alpha=gamma/(2.0*Ew_new);
709          beta=(n-gamma)/(2.0*Ed_new);
710          printf("gamma=%4.4f\talpha=%4.4f\tbeta=%4.4f\n",gamma,alpha,beta);
711                printf("Elapsed time inverse=%4.4f\n",t4-t3);
712          printf("#------------------------------------------------------------ #\n");
713          epoch++;
714 
715   }
```

```
716
717         free(X);
718         free(y);
719         free(yhat);
720         free(residuals);
721         free(J);
722         free(H);
723         free(Hinv);
724         free(theta);
725         free(theta_new);
726         free(delta);
727         free(g);
728         free(diag_H);
729         free(F_history);
730
731   return 0;
732 }
```

Anexo B: Código en *C/CUDA* con las bibliotecas *cuBLAS* y *cuSolver* versión 2

El siguiente código en el lenguaje de programación *C/CUDA* contiene la implementación del programa para la regularización Bayesiana de una red neuronal artificial. El código hace uso de las bibliotecas para álgebra lineal *cuBLAS* y *cuSolver* y traslada el cálculo de la matriz Jacobiana a la GPU para disminuir las transferencias de datos entre CPU y GPU.

```
#include <stdio.h>
//#include <stdlib.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#include <cusolverDn.h>


void print_array_column_major(double *in, int nrows, int ncolumns)
{
  int i,j;

  for(i=0; i<nrows;i++)
  {
    for(j=0; j<ncolumns;j++)
    {
      printf("%f \t",in[j*nrows+i]);
    }
    printf("\n");
  }
}

void welcome()
{
  printf("\n");
  printf("#________________________________________#\n");
  printf("# trainbr-CUDA                           #\n");
  printf("# Bayesian regularized neural networks   #\n");
  printf("#________________________________________#\n");
  printf("\n");
}

double cor_Pearson(double *x, double *y, int n)
{
   double sumx=0;
   double sumy=0;
   double sumxy=0;
   double sumx2=0;
   double sumy2=0;
   int i;

   for(i=0; i<n;i++)
   {
      sumx+=x[i];
      sumy+=y[i];
      sumxy+=x[i]*y[i];
      sumx2+=x[i]*x[i];
```

```
47        sumy2+=y[i]*y[i];
48      }
49      return((sumxy-sumx*sumy/(double (n)))/(pow((sumx2-sumx*sumx/(double (n)))*(sumy2-
        sumy*sumy/(double (n))),0.5)));
50  }
51
52  void read_matrix(double *X, int rows, int columns, char *Input_file)
53  {
54      const int MAX_LEN = 32767;
55      char Buffer[MAX_LEN];
56      char *token=NULL;
57      FILE *ptr;
58      int i,j;
59
60      ptr=fopen(Input_file,"r");
61      if(ptr!=NULL)
62      {
63        i=0;
64    printf("Loading␣incidence␣matrix...");
65          while(fgets(Buffer, MAX_LEN, ptr) != NULL)
66        {
67            token=strtok(Buffer,",");
68            j=0;
69            while(token!=NULL)
70      {
71              X[i+(j*rows)]=atof(token);
72              token = strtok(NULL,",");
73              j++;
74           }
75           i++;
76        }
77        printf("Done\n");
78          if(i!=rows)
79          {
80            printf("The␣file␣has␣MORE/LESS␣lines␣that␣those␣indicated␣by␣the␣user\n");
81            exit(1);
82          }
83          fclose(ptr);
84      }else{
85          printf("Unable␣to␣open␣input␣file␣with␣incidence␣matrix\n");
86    exit(1);
87      }
88  }
89
90  void read_vector(double *y, int rows,char *Input_file)
91  {
92        const int MAX_LEN=256;
93        char Buffer[MAX_LEN];
94        int i;
95        FILE *ptr;
96        ptr=fopen(Input_file,"r");
97        if(ptr!=NULL)
98        {
99    i=0;
```

```
100   printf("Loading␣response␣vector...");
101   while(fgets(Buffer, MAX_LEN, ptr) != NULL)
102   {
103       y[i]=atof(Buffer);
104       i++;
105       }
106       printf("Done\n");
107   if(i!=rows)
108         {
109       printf("The␣file␣has␣MORE/LESS␣lines␣that␣those␣indicated␣by␣the␣user\n");
110             exit(1);
111         }
112         fclose(ptr);
113       }else{
114     printf("Unable␣to␣open␣file␣with␣response␣vector\n");
115     exit(1);
116       }
117 }
118
119 __device__ double devtansig(double x)
120 {
121     return(2.0/(1.0+exp(-2.0*x)) - 1.0);
122 }
123
124
125 __device__ double sec1(double x)
126 {
127     return(2.0*exp(x)/(exp(2.0*x)+1.0));
128 }
129
130
131 double tansig(double x)
132 {
133     return(2.0/(1.0+exp(-2.0*x)) - 1.0);
134 }
135
136
137 double sech(double x)
138 {
139     return(2.0*exp(x)/(exp(2.0*x)+1.0));
140 }
141
142 __global__ void devpredictions_nn(double *X, unsigned int rows,
143                       unsigned int columns, double *theta,
144                       unsigned int neurons,double *yhat, double *y,
145                       double *residuals, double *ED)
146 {
147    unsigned int i,j,k;
148    double suma,z;
149    double error=0;
150
151    for(i=0;i<rows;i++)
152    {
153       suma=0;
```

```
154       for(k=0;k<neurons;k++)
155       {
156    z=0;
157    for(j=0;j<columns;j++)
158    {
159       z += X[i+(j*rows)] * theta[(columns+2)*k+j+2];  //1 vez por cada neurona
160    }
161    z += theta[(columns+2)*k+1];
162    suma += theta[(columns+2)*k]*devtansig(z);
163       }
164       yhat[i]=suma;
165       residuals[i] = y[i] - yhat[i];
166       error += pow(residuals[i],2.0);//pow(residuals[i],2.0);
167    }
168    *ED = error;
169 }
170
171 double predictions_nn(double *X, unsigned int rows,
172                       unsigned int columns, double *theta,
173                       unsigned int neurons,double *yhat, double *y,
174                       double *residuals)
175 {
176    unsigned int i,j,k;
177    double suma,z;
178    double error=0;
179
180    for(i=0;i<rows;i++)
181    {
182       suma=0;
183       for(k=0;k<neurons;k++)
184       {
185    z=0;
186    for(j=0;j<columns;j++)
187    {
188       z+=X[i+(j*rows)]*theta[(columns+2)*k+j+2];
189    }
190    z+=theta[(columns+2)*k+1];
191    suma+=theta[(columns+2)*k]*tansig(z);
192       }
193       yhat[i]=suma;
194       residuals[i]=y[i]-yhat[i];
195       error+=pow(residuals[i],2.0);
196    }
197    return(error);
198 }
199
200
201
202 __global__ void devpredictions1_nn(double *X, unsigned int rows,
203                       unsigned int columns, double *theta,
204                       unsigned int neurons,double *yhat)
205 {
206    unsigned int i,j,k;
207    double suma,z;
```

```
208
209     for(i=0;i<rows;i++)
210     {
211        suma=0;
212        for(k=0;k<neurons;k++)
213            {
214            z=0;
215            for(j=0;j<columns;j++)
216                {
217                z = z + X[i+(j*rows)] * theta[(columns+2) * k + j + 2];
218                }
219            z = z + theta[(columns+2) * k + 1];
220            suma = suma + theta[(columns+2)*k]*devtansig(z);
221             }
222        yhat[i] = suma;
223     }
224          //printf("\npredictions done\n");
225
226 }
227
228
229
230 void predictions_nn(double *X, unsigned int rows,
231                     unsigned int columns, double *theta,
232                     unsigned int neurons,double *yhat)
233 {
234     unsigned int i,j,k;
235     double suma,z;
236
237     for(i=0;i<rows;i++)
238     {
239        suma=0;
240        for(k=0;k<neurons;k++)
241        {
242     z=0;
243     for(j=0;j<columns;j++)
244     {
245        z+=X[i+(j*rows)]*theta[(columns+2)*k+j+2];
246     }
247     z+=theta[(columns+2)*k+1];
248     suma+=theta[(columns+2)*k]*tansig(z);
249        }
250        yhat[i]=suma;
251     }
252 }
253
254 __global__ void jacobian(double *X, unsigned int n,
255                 unsigned int p, double *theta,
256                 unsigned int neurons,double *J)
257 {
258     unsigned int i,j,k;
259     double z,dtansig, sec;
260
261     //printf("Calulating jacobian...");
```

```
262    for(i=0; i<n; i++)
263    {
264      for(k=0; k<neurons; k++)
265      {
266     z=0;
267         for(j=0;j<p;j++)
268         {
269          z = z + X[i+(j * n)] * theta[(p+2)*k+j+2]; //* k + j + 2 sin parentesis
270         }
271     z = z + theta[(p+2) * k + 1];
272       sec = sec1(z);
273     dtansig = sec * sec; //pow(sec,2.0);
274
275     J[i+(((p+2) * k) * n)] = -1.0 * devtansig(z);
276
277     J[i+(((p+2) * k+1) * n)]= -1.0 * theta[(p+2) * k] * dtansig;
278
279    for(j=0; j<p;j++)
280    {
281        J[i+(((p+2) * k + j + 2) * n)]= -1.0 * theta[(p+2) * k] * dtansig * X[i+(j * n)
       ];
282    }
283      }
284    }
285    //printf("Done\n");
286 }
287
288
289 __global__ void devsc(double *theta, int npar, double *EW)
290 {
291    double suma=0, teta_i;
292    for(int i=0; i<npar; i++)
293         {
294         teta_i = theta[i];
295         suma = suma + (teta_i * teta_i); //pow(theta[i],2.0);
296         }
297    *EW = suma;
298
299         printf("\nEw␣calculado␣=␣%f\n", *EW);
300
301 }
302
303
304 __global__ void inicialbeta(double *beta, int n, double *gamma, double *Ed)
305 {
306         double bt = 0.0, gmm, errord;
307         gmm = *gamma;
308         errord = *Ed;
309         bt = (n - (gmm))/(2.0 * errord);
310         if(bt < 0) bt = 1.0;
311         *beta = bt;
312 }
313 /*****************************************************************************/
314
```

```
315 __global__ void inicialalpha(double *alpha, double *gamma, double *Ew)
316 {
317         double alfa=0.0, gmm, sumaw;
318         //alfa = *alpha;
319         sumaw = *Ew;
320         gmm = *gamma;
321         alfa = gmm/(2.0 * sumaw);
322         *alpha = alfa;
323 }
324
325 /*****************************************************************************/
326
327 __global__ void imprime_ab(double *alpha, double *beta)
328 {
329
330        printf("alpha=%4.4f\tbeta=%4.4f\n",*alpha,*beta);
331 }
332
333 /*****************************************************************************/
334
335 __global__ void imprimeDW(double *Ed, double *Ew)
336 {
337
338        printf("Ed=%4.4f\tEw=%4.4f\n",(*Ed),(*Ew));
339
340 }
341
342 /*****************************************************************************/
343
344
345 __global__ void calc_GDH(double *g, int numpar, double *alpha, double *beta,
346                          double *theta, double *Hessian, double *diagHessian)
347 {
348     int i;
349     double alfa, bt;
350         alfa = *alpha;
351         bt = *beta;
352
353        for(i = 0; i < numpar; i++)
354        {
355           g[i] = g[i] + ((alfa / bt) * theta[i]);
356           //printf("i = %d = %f\n",i, g[i]);
357           diagHessian[i] = Hessian[i+(i * numpar)];
358           //printf("diagH i = %d = %f\n",i, diagHessian[i]);
359
360        }
361         printf("\nCalc␣GDH␣done\n");
362
363 }
364
365
366 /*****************************************************************************/
367
368 __global__ void calc_C(double *C, double *beta, double *Ed,
```

```
369                         double *alpha, double *Ew, int *flag_C )//saque flag_C
370 {
371             double bt, errord, alfa, sumaw, ce = 0.0;
372             bt = *beta;
373             errord = *Ed;
374             alfa = *alpha;
375             sumaw= *Ew;
376 
377             ce = bt*errord + alfa*sumaw;
378         *C = ce;
379             *flag_C = 1;
380 
381         printf("\nCalc␣C␣done,␣C␣=␣%f\n", *C);
382 
383 }
384 
385 
386 /*********************************************************************/
387 
388 __global__ void calc_H_delta(double *H, double *diagonal, double *alpha, double *beta,
      double *mu,
389                                 double *delta, double *g, int npar)
390 {
391             int i;
392             double alfa, bt, MU;
393             alfa = *alpha;
394             bt = *beta;
395             MU = *mu;
396 
397         for(i=0; i<npar;i++)
398               {
399              H[i+(i * npar)] = diagonal[i]+(alfa/bt)+(MU/(2.0*bt));
400              delta[i]=g[i];
401                 }
402         printf("\nCalc␣H␣delta␣done\n");
403 
404 }
405 
406 /*********************************************************************/
407 
408 __global__ void update_H(double *H, double *diagonal, double *beta, double *alpha, int
      npar)
409 {
410         int i, j;
411         double alfa, bt;
412         alfa = *alpha;
413         bt = *beta;
414 
415       for(i=0; i<npar;i++)
416           {
417           for(j=0; j<npar;j++)
418               {
419               if(i==j)
420                     {
```

```
421                 H[i+(i * npar)] = 2.0 * bt * diagonal[i] + 2.0 * alfa;
422                   }
423                 else
424                     {
425                  H[i+(j * npar)] = 2.0 * bt * H[i+(j * npar)];
426                   }
427               }
428           }
429
430         printf("\nupdate␣H␣done\n");
431
432 }
433 /*************************************************************/
434
435 __global__ void updateThetaTrace(double *theta, double *theta_new, double *trace, int
      npar, double *Hinv)
436 {
437         int i;
438         double traza = 0.0;
439
440       for(i=0; i<npar; i++)
441           {
442              theta[i] = theta_new[i];
443              traza += Hinv[i+(i * npar)];
444           }
445         *trace = traza;
446 }
447
448 __global__ void lastblock(double *gamma, double *alpha, double *beta, double *Ew_new,
      double *Ed_new,
449                               double *trace, int n, int npar, int *epoch)
450 {
451         double gmm, alfa, bt, traza, nuevoew, ednuevo;
452
453         //gmm = *gamma;
454         alfa = *alpha;
455         bt = *beta;
456         traza = *trace;
457         nuevoew = *Ew_new;
458         ednuevo = *Ed_new;
459       gmm = npar - (2.0 * alfa * traza);
460       alfa = gmm / (2.0 * nuevoew);
461       bt = (n - gmm) / (2.0 * ednuevo);
462         *alpha = alfa;
463         *beta = bt;
464         *gamma = gmm;
465       printf("gamma=%4.4f\talpha=%4.4f\tbeta=%4.4f\n",*gamma,*alpha,*beta);
466             //printf("Elapsed time inverse=%4.4f\n",t4-t3);
467       printf("#-------------------------------------------------------␣#\n");
468
469       *epoch = *epoch + 1;
470
471 }
472
```

```
473
474 void initnw(double *theta, int n, int p, int neurons, int npar)
475 {
476   double scaling_factor,dx, suma;
477   int k,j;
478
479   srand(time(NULL));
480   for(int i=0; i<npar;i++)
481   theta[i]=((double) rand()) / ((double) RAND_MAX)-0.5;
482
483   printf("Initializing␣using␣the␣Nguyen-Widrow␣method\n");
484   if(p==1)
485   {
486       scaling_factor=0.7*neurons;
487       for(k=0;k<neurons;k++)
488       {
489     theta[(p+2)*k+1]=scaling_factor*(1.0-2.0*(((double) rand())
490          / ((double) RAND_MAX)));
491     theta[(p+2)*k+2]=scaling_factor;
492       }
493   }else{
494      scaling_factor=0.7*pow(neurons,(1.0/double (n)));
495      dx=2.0/double(neurons);
496      for(k=0; k<neurons;k++)
497      {
498           suma=0;
499     for(j=0;j<p;j++)
500     {
501       suma+=pow(theta[(p+2)*k+2+j],2.0);
502     }
503     for(j=0; j<p; j++)
504     {
505       theta[(p+2)*k+2+j]=scaling_factor*theta[(p+2)*k+2+j]/pow(suma,0.5);
506     }
507     theta[(p+2)*k+1]=(-1.0+dx*k)*scaling_factor;
508      }
509   }
510 }
511
512 /*
513  * Crossprod of a matrix using cublas
514  * C=A'A
515  */
516
517 void matrix_crossprod(double *A, int row_a, int col_a, double *C, int row_c, int col_c
      )
518 {
519     double alpha=1.0;
520     double beta=0.0;
521
522     /*cuda malloc status*/
523     cudaError_t cudaStat;
524
525     /*cublas functions status*/
```

```
526     cublasStatus_t stat;
527 
528     /*cublas handle status*/
529     cublasHandle_t handle;
530 
531     /*Memory allocation in the device*/
532     double *d_A;
533     double *d_C;
534 
535     cudaStat=cudaMalloc((void**)&d_A,row_a*col_a*sizeof(double)); //device memory
        allocation for d_A
536     cudaStat=cudaMalloc((void**)&d_C,row_c*col_c*sizeof(double)); //device memory
        allocation for d_C
537 
538     stat = cublasCreate(&handle); // initialize CUBLAS context
539 
540     cudaMemcpy(d_A,A,row_a*col_a*sizeof(double),cudaMemcpyHostToDevice);
541     cudaMemcpy(d_C,C,row_c*col_c*sizeof(double),cudaMemcpyHostToDevice);
542 
543      stat=cublasDgemm(handle,
544                       CUBLAS_OP_T,
545                       CUBLAS_OP_N,
546                       col_a,
547                       col_a,
548                       row_a,
549                       &alpha,
550                       d_A,
551                       row_a,
552                       d_A,
553                       row_a,
554                       &beta,
555                       d_C,
556                       col_a
557                       );
558 
559     if(stat!=CUBLAS_STATUS_SUCCESS)
560     {
561   printf("Error in cublas\n");
562   exit(1);
563     }
564 
565     cudaMemcpy(C,d_C,row_c*col_c*sizeof(double),cudaMemcpyDeviceToHost);
566 
567     /*Free memory*/
568     cudaFree(d_A);
569     cudaFree(d_C);
570     cublasDestroy(handle);
571 }
572 
573 /*
574  * Matrix vector product using cublas
575  * sol=A'b
576  */
577 void matrix_vector_product(double *A, int row_a, int col_a, double *b,  double *sol)
```

```
578 {
579     int one=1;
580     double alpha=1.0;
581     double beta=0.0;
582 
583     /*cuda malloc status*/
584     cudaError_t cudaStat;
585 
586     /*cublas functions status*/
587     cublasStatus_t stat;
588 
589     /*cublas handle status*/
590     cublasHandle_t handle;
591 
592     /*Memory allocation in the device*/
593     double *d_A;
594     double *d_b;
595     double *d_sol;
596 
597     cudaStat=cudaMalloc((void**)&d_A,row_a*col_a*sizeof(double)); //device memory
      allocation for d_A
598     cudaStat=cudaMalloc((void**)&d_b,row_a*sizeof(double));       //device memory
      allocation for d_b
599     cudaStat=cudaMalloc((void**)&d_sol,col_a*sizeof(double));   //device memory
      allocation for d_sol
600 
601     stat = cublasCreate(&handle); // initialize CUBLAS context
602 
603     cudaMemcpy(d_A,A,row_a*col_a*sizeof(double),cudaMemcpyHostToDevice);
604     cudaMemcpy(d_b,b,row_a*sizeof(double),cudaMemcpyHostToDevice);
605 
606     stat=cublasDgemv(handle,
607                      CUBLAS_OP_T,
608                      row_a,
609                      col_a,
610                      &alpha,
611                      d_A,
612                      row_a,
613                      d_b,
614                      one,
615                      &beta,
616                      d_sol,
617                      one);
618 
619     cudaMemcpy(sol,d_sol,col_a*sizeof(double),cudaMemcpyDeviceToHost);
620 
621     /*Free memory*/
622     cudaFree(d_A);
623     cudaFree(d_b);
624     cudaFree(d_sol);
625     cublasDestroy(handle);
626 }
627 
628 /*
```

```
629 Solve a linear system of equations using cuda Ax=b,
630 The solution x, will be given in vector b, so the original one will be lost
631 */
632
633 void solve_Cholesky(double *A, int row_a, int col_a, double *b)
634 {
635    cudaError cudaStatus;
636    cusolverStatus_t cusolverStatus;
637    cusolverDnHandle_t handle;
638
639    double *d_A, *d_b, *Work; // matrix A, rhs b and worksp.
640    int *d_info, Lwork; // device version of info, worksp.size
641    int info_gpu = 0; // device info copied to host
642
643    cudaStatus = cudaGetDevice(0);
644    cusolverStatus = cusolverDnCreate(&handle); // create handle
645    cublasFillMode_t uplo = CUBLAS_FILL_MODE_LOWER;
646
647    // prepare memory on the device
648    cudaStatus = cudaMalloc((void**)&d_A, row_a*row_a*sizeof(double));
649    cudaStatus = cudaMalloc((void**)&d_b, row_a*sizeof(double));
650    cudaStatus = cudaMalloc((void**)&d_info, sizeof(int));
651    cudaStatus = cudaMemcpy(d_A, A, row_a*row_a*sizeof(double),cudaMemcpyHostToDevice);
       // copy A->d_A
652    cudaStatus = cudaMemcpy(d_b, b, row_a*sizeof(double),cudaMemcpyHostToDevice); //
      copy b->d_b
653
654    // compute workspace size and prepare workspace
655    cusolverStatus = cusolverDnDpotrf_bufferSize(handle, uplo, row_a, d_A,row_a, &Lwork
      );
656
657    cudaStatus = cudaMalloc((void**)&Work,Lwork*sizeof(double));
658
659    // Cholesky decomposition d_A=L*L^T, lower triangle of d_A is
660    // replaced by the factor L
661    cusolverStatus = cusolverDnDpotrf(handle,uplo,row_a,d_A,row_a,Work, Lwork,d_info);
662
663
664    if(cusolverStatus==CUSOLVER_STATUS_SUCCESS)
665    {
666      printf("Successful␣Cholesky␣factorization\n");
667    }
668
669    // solve d_A*x=d_b, where d_A is factorized by potrf function
670    // d_b is overwritten by the solution
671    cusolverStatus = cusolverDnDpotrs(handle,uplo,row_a, 1,d_A,row_a, d_b,row_a,d_info)
      ;
672 /*
673    if(cusolverStatus==CUSOLVER_STATUS_SUCCESS)
674    {
675       printf("Successful solution\n");
676    }
677 */
678    cudaStatus = cudaDeviceSynchronize();
```

```
679
680     cudaStatus = cudaMemcpy(b, d_b, row_a*sizeof(double),cudaMemcpyDeviceToHost ); //
         copy solution to host d_b ->b
681
682     // free memory
683     cudaFree(d_A);
684     cudaFree(d_b);
685     cudaFree(d_info);
686     cudaFree(Work);
687     cusolverStatus = cusolverDnDestroy(handle);
688     //cudaStatus = cudaDeviceReset();
689
690 }
691
692 /*
693  * Invert a matrix using cuda
694  * WARNING: Only the UPPER part is useful
695  */
696
697 void inverse(double *A, int row_a, double *Ainv)
698 {
699
700     cudaError cudaStatus;
701     cusolverStatus_t cusolverStatus;
702     cusolverDnHandle_t handle;
703
704     double *d_A, *d_Ainv, *Work; // matrix A and worksp.
705     int *d_info, Lwork; // device version of info, worksp.size
706     int info_gpu = 0; // device info copied to host
707
708     cudaStatus = cudaGetDevice(0);
709     cusolverStatus = cusolverDnCreate(&handle); // create handle
710     cublasFillMode_t uplo = CUBLAS_FILL_MODE_LOWER;
711
712     // prepare memory on the device
713     cudaStatus = cudaMalloc((void**)&d_A, row_a*row_a*sizeof(double));
714     cudaStatus = cudaMalloc((void**)&d_Ainv,row_a*row_a*sizeof(double));
715     cudaStatus = cudaMalloc((void**)&d_info, sizeof(int));
716     cudaStatus = cudaMemcpy(d_A, A, row_a*row_a*sizeof(double),cudaMemcpyHostToDevice);
          // copy A->d_A
717     cudaStatus = cudaMemcpy(d_Ainv,Ainv,row_a*row_a*sizeof(double),
         cudaMemcpyHostToDevice); // copy Ainv->d_Ainv
718
719     // compute workspace size and prepare workspace
720     cusolverStatus = cusolverDnDpotrf_bufferSize(handle, uplo, row_a, d_A,row_a, &Lwork
         );
721
722     cudaStatus = cudaMalloc((void**)&Work,Lwork*sizeof(double));
723
724     // Cholesky decomposition d_A=L*L^T, lower triangle of d_A is
725     // replaced by the factor L
726     cusolverStatus = cusolverDnDpotrf(handle,uplo,row_a,d_A,row_a,Work, Lwork,d_info);
727
728     // solve d_A*x=d_b, where d_A is factorized by potrf function
```

```
729     // d_Ainv is overwritten by the solution
730     cusolverStatus = cusolverDnDpotrs(handle,uplo,row_a, row_a,d_A,row_a, d_Ainv,row_a,
        d_info);
731
732     cudaStatus = cudaDeviceSynchronize();
733
734     cudaStatus = cudaMemcpy(Ainv, d_Ainv, row_a*row_a*sizeof(double),
        cudaMemcpyDeviceToHost ); // copy solution to host d_Ainv -> Ainv
735
736     // free memory
737     cudaFree(d_A);
738     cudaFree(d_Ainv);
739     cudaFree(d_info);
740     cudaFree(Work);
741     cusolverStatus = cusolverDnDestroy(handle);
742     //cudaStatus = cudaDeviceReset();
743
744 }
745
746 void inverse2(double *A, int row_a)
747 {
748
749     cudaError cudaStatus;
750     cusolverStatus_t cusolverStatus;
751     cusolverDnHandle_t handle;
752
753     double *d_A, *Work; // matrix A and worksp.
754     int *d_info, Lwork; // device version of info, worksp.size
755     int info_gpu = 0; // device info copied to host
756
757     cudaStatus = cudaGetDevice(0);
758     cusolverStatus = cusolverDnCreate(&handle); // create handle
759     cublasFillMode_t uplo = CUBLAS_FILL_MODE_LOWER;
760
761     // prepare memory on the device
762     cudaStatus = cudaMalloc((void**)&d_A, row_a*row_a*sizeof(double));
763     cudaStatus = cudaMalloc((void**)&d_info, sizeof(int));
764     cudaStatus = cudaMemcpy(d_A, A, row_a*row_a*sizeof(double),cudaMemcpyHostToDevice);
          // copy A->d_A
765
766     // compute workspace size and prepare workspace
767     cusolverStatus = cusolverDnDpotrf_bufferSize(handle, uplo, row_a, d_A,row_a, &Lwork
        );
768
769     cudaStatus = cudaMalloc((void**)&Work,Lwork*sizeof(double));
770
771     // Cholesky decomposition d_A=L*L^T, lower triangle of d_A is
772     // replaced by the factor L
773     cusolverStatus = cusolverDnDpotrf(handle,uplo,row_a,d_A,row_a,Work, Lwork,d_info);
774
775     cudaStatus = cudaDeviceSynchronize();
776
777     cudaStatus = cudaMemcpy(A, d_A, row_a*row_a*sizeof(double),cudaMemcpyDeviceToHost )
        ; // copy solution to host d_Ainv -> Ainv
```

```
778
779     // free memory
780     cudaFree(d_A);
781     cudaFree(d_info);
782     cudaFree(Work);
783     cusolverStatus = cusolverDnDestroy(handle);
784     cudaStatus = cudaDeviceReset();
785
786 }
787
788
789 int main(int argc, char **argv)
790 {
791
792         cudaDeviceReset();
793
794   int i,j;
795         double *X, *y,*yhat, *residuals,*theta, *theta_new,
796                  *J, *H, *delta, *Hinv, *g, *diag_H, *F_history;
797
798 /*******************************************************/
799   double *gamma;
800         double *Ed, *Ed_new;
801         double *Ew, *Ew_new;
802         double *alpha;
803         double *beta;
804      double *C;
805         double *C_new;
806
807         int *epoch;//=0;
808         int *max_epochs;//=1000;
809         double *mu;//=0.005;
810         double *mu_inc;//=10.0;
811         double *mu_dec;//=0.1;
812         double *mu_max;//=1e10;
813         double *change;//=0.001;
814
815         double *trace;
816 /*******************************************************/
817
818
819   /*
820   Read command line arguments, pass 1
821   */
822   int n=atoi(argv[1]);
823         int p=atoi(argv[2]);
824         int neurons=atoi(argv[3]);
825   int npar=neurons*(2+p);
826
827   /*
828   End of reading command line arguments, pass 1
829   */
830
831   /*
```

```
832   Initialization
833   */
834
835   welcome();
836
837   printf("Cases: %d\n",n);
838         printf("Predictors: %d\n",p);
839   printf("Neurons: %d\n",neurons);
840   printf("Number of parameters to estimate: %d\n",npar);
841
842 /*************************************************************/
843       gamma = (double*)malloc(sizeof(double));
844       Ed = (double*)malloc(sizeof(double));
845         Ed_new = (double*)malloc(sizeof(double));
846       Ew = (double*)malloc(sizeof(double));
847         Ew_new = (double*)malloc(sizeof(double));
848       alpha = (double*)malloc(sizeof(double));
849       beta = (double*)malloc(sizeof(double));
850         C = (double*)malloc(sizeof(double));
851       C_new = (double*)malloc(sizeof(double));
852
853       epoch = (int*)malloc(sizeof(int));//=0;
854       max_epochs = (int*)malloc(sizeof(int));//=1000;
855       mu = (double*)malloc(sizeof(double));//=0.005;
856       mu_inc = (double*)malloc(sizeof(double));//=10.0;
857       mu_dec = (double*)malloc(sizeof(double));//=0.1;
858       mu_max = (double*)malloc(sizeof(double));//=1e10;
859       change = (double*)malloc(sizeof(double));//=0.001;
860
861         trace = (double*)malloc(sizeof(double));
862
863       *epoch=0;
864       *max_epochs=1000;
865       *mu=0.005;
866       *mu_inc=10.0;
867       *mu_dec=0.1;
868       *mu_max=1e10;
869       *change=0.001;
870 /*************************************************************/
871
872
873   X = (double *) malloc(n*p*sizeof(double));
874   y = (double *) malloc(n*sizeof(double));
875   yhat=(double *) malloc(n*sizeof(double));
876   residuals=(double *) malloc(n*sizeof(double));
877   J=(double *) malloc(n*npar*sizeof(double));
878   H=(double *) malloc(npar*npar*sizeof(double));
879   Hinv=(double *) malloc(npar*npar*sizeof(double));
880   theta=(double *) malloc(npar*sizeof(double));
881   theta_new=(double *) malloc(npar*sizeof(double));
882   delta=(double *) malloc(npar*sizeof(double));
883   g=(double *) malloc(npar*sizeof(double));
884   diag_H=(double *) malloc(npar*sizeof(double));
885   F_history=(double *) malloc(*max_epochs * sizeof(double));
```

```
886
887   read_matrix(X,n,p,argv[4]);
888   read_vector(y,n,argv[5]);
889
890   initnw(theta,n,p,neurons,npar);
891 /*****************************************************************/
892   cudaError_t cudaStat;
893     /*Memory allocation in the device*/
894     double *d_1X;
895     double *d_1J;
896     double *d_1theta;
897     double *d_y, *d_yhat, *d_residuals, *d_Ew, *d_Ed;
898     double *d_alpha, *d_beta, *d_gamma;
899     double *d_g, *d_H, *d_diag_H, *d_Hinv;
900     double *d_C, *d_theta_new, *d_trace, *d_Ew_new, *d_Ed_new;
901     double *d_mu, *d_delta;
902     int *d_flag_C, *d_epoch;
903
904
905
906     cudaStat=cudaMalloc((void**)&d_1X,n*p*sizeof(double));
907     cudaStat=cudaMalloc((void**)&d_1J,n*npar*sizeof(double));
908   cudaStat = cudaMalloc((void**)&d_1theta,npar*sizeof(double));
909   cudaStat = cudaMalloc((void**)&d_y,n*sizeof(double));
910   cudaStat = cudaMalloc((void**)&d_yhat,n*sizeof(double));
911   cudaStat = cudaMalloc((void**)&d_residuals,n*sizeof(double));
912     cudaStat = cudaMalloc((void**)&d_Ed, sizeof(double));
913     cudaStat = cudaMalloc((void**)&d_Ew, sizeof(double));
914     cudaStat = cudaMalloc((void**)&d_alpha, sizeof(double));
915     cudaStat = cudaMalloc((void**)&d_beta, sizeof(double));
916   cudaStat = cudaMalloc((void**)&d_gamma, sizeof(double));
917   cudaStat = cudaMalloc((void**)&d_H,npar*npar*sizeof(double));
918   cudaStat = cudaMalloc((void**)&d_Hinv,npar*npar*sizeof(double));
919   cudaStat = cudaMalloc((void**)&d_g,npar*sizeof(double));
920   cudaStat = cudaMalloc((void**)&d_diag_H,npar*sizeof(double));
921     cudaStat = cudaMalloc((void**)&d_C, sizeof(double));
922     cudaStat = cudaMalloc((void**)&d_flag_C, sizeof(int));
923   cudaStat = cudaMalloc((void**)&d_theta_new,npar*sizeof(double));
924     cudaStat = cudaMalloc((void**)&d_trace, sizeof(double));
925   cudaStat = cudaMalloc((void**)&d_delta,npar*sizeof(double));
926     cudaStat = cudaMalloc((void**)&d_mu, sizeof(double));
927
928     cudaMemcpy(d_1X,X,n*p*sizeof(double),cudaMemcpyHostToDevice);
929     cudaMemcpy(d_1theta,theta,npar*sizeof(double),cudaMemcpyHostToDevice);
930     cudaMemcpy(d_y, y, n*sizeof(double), cudaMemcpyHostToDevice);
931     cudaMemcpy(d_yhat, yhat, n*sizeof(double), cudaMemcpyHostToDevice);
932
933     cudaMemcpy(d_alpha, alpha, sizeof(double), cudaMemcpyHostToDevice);
934     cudaMemcpy(d_beta, beta, sizeof(double), cudaMemcpyHostToDevice);
935     cudaMemcpy(d_gamma, gamma, sizeof(double), cudaMemcpyHostToDevice);
936
937     cudaStat = cudaMalloc((void**)&d_epoch, sizeof(int));
938     cudaStat = cudaMalloc((void**)&d_Ed_new, sizeof(double));
939     cudaStat = cudaMalloc((void**)&d_Ew_new, sizeof(double));
```

```
940
941     cudaMemcpy(d_epoch, epoch, sizeof(int), cudaMemcpyHostToDevice);
942
943     cudaStat = cudaDeviceSynchronize();
944
945
946 /*****************************************************************************/
947
948       *gamma =  npar;
949         cudaStat = cudaMemcpy(d_gamma, gamma, sizeof(double),cudaMemcpyHostToDevice);
950         cudaStat = cudaDeviceSynchronize();
951
952 /*****************************************************************************/
953
954         devpredictions_nn<<<1,1>>>(d_1X, n, p, d_1theta, neurons, d_yhat, d_y,
      d_residuals, d_Ed);
955
956         cudaStat = cudaDeviceSynchronize();
957
958         inicialbeta<<<1,1>>>(d_beta, n, d_gamma, d_Ed);
959         cudaStat = cudaDeviceSynchronize();
960
961         devsc<<<1,1>>>(d_1theta,npar, d_Ew);
962         cudaStat = cudaDeviceSynchronize();
963
964         inicialalpha<<<1,1>>>(d_alpha, d_gamma, d_Ew);
965         cudaStat = cudaDeviceSynchronize();
966
967
968     cudaMemcpy(alpha, d_alpha, sizeof(double), cudaMemcpyDeviceToHost);
969     cudaMemcpy(beta, d_beta, sizeof(double), cudaMemcpyDeviceToHost);
970     cudaMemcpy(gamma, d_gamma, sizeof(double), cudaMemcpyDeviceToHost);
971     cudaMemcpy(Ed, d_Ed, sizeof(double), cudaMemcpyDeviceToHost);
972     cudaMemcpy(Ew, d_Ew, sizeof(double), cudaMemcpyDeviceToHost);
973     cudaMemcpy(yhat, d_yhat, n*sizeof(double), cudaMemcpyDeviceToHost);
974     cudaMemcpy(residuals, d_residuals, n*sizeof(double), cudaMemcpyDeviceToHost);
975     cudaStat = cudaDeviceSynchronize();
976
977   /*
978   End of initialization
979   */
980
981   /*
982   main loop
983   */
984
985       int *flag_C;
986       int flag_mu=1;
987       int flag_change_F=1;
988
989       flag_C = (int*)malloc(sizeof(int));
990
991
992
```

```
993       while(*epoch < *max_epochs && flag_mu && flag_change_F)
994   {
995        printf("epoch=%d\n",*epoch);
996        printf("alpha=%4.4f\tbeta=%4.4f\n",*alpha,*beta);
997 /*****************************************************************************/
998 /*****************************************************************************/
999     cudaMemcpy(d_1theta,theta,npar*sizeof(double),cudaMemcpyHostToDevice);
1000
1001
1002        /*Calculate the Jacobian*/
1003              jacobian<<<1,1>>>(d_1X,n,p,d_1theta,neurons,d_1J);
1004         cudaStat = cudaDeviceSynchronize();
1005
1006     cudaStat = cudaMemcpy(J,d_1J,n*npar*sizeof(double),cudaMemcpyDeviceToHost);
1007         cudaStat = cudaDeviceSynchronize();
1008
1009         cudaStat = cudaDeviceSynchronize();
1010 /*****************************************************************************/
1011 /*****************************************************************************/
1012
1013        /*Calculate the residuals*/
1014
1015     cudaMemcpy(d_alpha, alpha, sizeof(double), cudaMemcpyHostToDevice);
1016     cudaMemcpy(d_beta, beta, sizeof(double), cudaMemcpyHostToDevice);
1017     cudaMemcpy(d_gamma, gamma, sizeof(double), cudaMemcpyHostToDevice);
1018         cudaStat = cudaDeviceSynchronize();
1019
1020         devpredictions_nn<<<1,1>>>(d_1X, n, p, d_1theta, neurons, d_yhat, d_y,
      d_residuals, d_Ed);
1021         devsc<<<1,1>>>(d_1theta,npar, d_Ew);
1022         cudaStat = cudaDeviceSynchronize();
1023
1024     cudaMemcpy(Ed, d_Ed, sizeof(double), cudaMemcpyDeviceToHost);
1025     cudaMemcpy(Ew, d_Ew, sizeof(double), cudaMemcpyDeviceToHost);
1026     cudaMemcpy(yhat, d_yhat, n*sizeof(double), cudaMemcpyDeviceToHost);
1027     cudaMemcpy(residuals, d_residuals, n*sizeof(double), cudaMemcpyDeviceToHost);
1028         cudaStat = cudaDeviceSynchronize();
1029
1030
1031 /*****************************************************************/
1032 /*****************************************************************/
1033
1034        /*
1035              Calculate the Hessian
1036              */
1037              memset(Hinv,0,npar*npar*sizeof(double));
1038              matrix_crossprod(J,n,npar,H, npar, npar);
1039
1040        /*
1041               Calculate the gradient
1042               Extract the diagonal of H
1043               Compute the cost
1044              */
1045
```

```
1046              matrix_vector_product(J,n,npar, residuals, g);
1047
1048 /*********************************************************************/
1049 /*********************************************************************/
1050     /*
1051     copy g, H and diagH
1052     */
1053
1054   cudaMemcpy(d_H, H, npar*npar*sizeof(double), cudaMemcpyHostToDevice);
1055   cudaMemcpy(d_g, g, npar*sizeof(double), cudaMemcpyHostToDevice);
1056   cudaMemcpy(d_diag_H, diag_H, npar*sizeof(double), cudaMemcpyHostToDevice);
1057
1058     cudaStat = cudaDeviceSynchronize();
1059
1060
1061     calc_GDH<<<1,1>>>(d_g, npar, d_alpha, d_beta, d_1theta, d_H, d_diag_H);
1062     cudaStat = cudaDeviceSynchronize();
1063
1064     cudaMemcpy(g, d_g, npar*sizeof(double), cudaMemcpyDeviceToHost);
1065     cudaMemcpy(diag_H, d_diag_H, npar*sizeof(double), cudaMemcpyDeviceToHost);
1066     cudaStat = cudaDeviceSynchronize();
1067
1068
1069 /*********************************************************************/
1070 /*********************************************************************/
1071
1072     calc_C<<<1,1>>>(d_C, d_beta, d_Ed, d_alpha, d_Ew, d_flag_C);
1073     cudaStat = cudaDeviceSynchronize();
1074
1075     cudaMemcpy(flag_C, d_flag_C, sizeof(int),cudaMemcpyDeviceToHost);
1076     cudaMemcpy(C, d_C, sizeof(double),cudaMemcpyDeviceToHost);
1077     cudaStat = cudaDeviceSynchronize();
1078
1079 /*********************************************************************/
1080 /*********************************************************************/
1081     while(*flag_C && flag_mu )
1082         {
1083         cudaMemcpy(d_mu, mu, sizeof(double), cudaMemcpyHostToDevice);
1084         cudaStat = cudaDeviceSynchronize();
1085
1086         calc_H_delta<<<1,1>>>(d_H, d_diag_H, d_alpha, d_beta, d_mu, d_delta, d_g, npar
     );
1087         cudaStat = cudaDeviceSynchronize();
1088
1089         cudaMemcpy(mu, d_mu, sizeof(double), cudaMemcpyDeviceToHost);
1090         cudaMemcpy(delta, d_delta, npar*sizeof(double),cudaMemcpyDeviceToHost);
1091         cudaMemcpy(H, d_H, npar*npar*sizeof(double), cudaMemcpyDeviceToHost);
1092         cudaStat = cudaDeviceSynchronize();
1093
1094 /*********************************************************************/
1095
1096       solve_Cholesky(H,npar,npar,delta);
1097
1098 /*********************************************************************/
```

```
1099
1100        for(i=0; i<npar;i++)
1101          {
1102            theta_new[i]=theta[i]-delta[i];
1103          }
1104
1105        *Ed_new=predictions_nn(X, n, p, theta_new,neurons,yhat,y,residuals);
1106        *Ew_new=sc(theta_new,npar);
1107        *C_new= *beta * *Ed_new + *alpha * *Ew_new;
1108        printf("C_new=%f\tmu=%f\n",*C_new,*mu);
1109
1110        if(*C_new < *C)
1111            {
1112              printf("C_new␣smaller␣than␣C\n");
1113            *mu = *mu * *mu_dec;
1114            if (*mu < 1e-20) *mu = 1e-20;
1115            *flag_C=0;
1116            F_history[*epoch] = *C_new;
1117            if(*epoch>3)
1118                {
1119                if((fabs(F_history[*epoch]-F_history[*epoch-1])< *change) && (fabs(
    F_history[*epoch-1]-F_history[*epoch-2])< *change)  && (fabs(F_history[*epoch-2]-
    F_history[*epoch-3])< *change))
1120                    {
1121                  flag_change_F=0;
1122                  printf("Changes␣in␣F=␣beta*SCE␣+␣alpha*Ew␣in␣last␣3␣iterations␣less␣
    than␣0.001\n");
1123                    }
1124                }
1125              }else{
1126                  printf("C_new␣bigger␣than␣C\n");
1127                *mu = *mu * *mu_inc;
1128                if(*mu > *mu_max)
1129                    {
1130                    flag_mu=0;
1131                    }
1132                }
1133       } //End of while
1134
1135             /*
1136              * Update all
1137             */
1138
1139        update_H<<<1,1>>>(d_H, d_diag_H, d_beta, d_alpha, npar);
1140
1141        cudaStat = cudaDeviceSynchronize();
1142        cudaMemcpy(H, d_H, npar*npar*sizeof(double), cudaMemcpyDeviceToHost);
1143        cudaStat = cudaDeviceSynchronize();
1144
1145    /*****************************************************************/
1146        //set Hinv to  identity
1147        memset(Hinv,0,npar*npar*sizeof(double));
1148        for(i=0;i<npar;i++)
1149            {
```

```
1150         Hinv[i+(i*npar)]=1.0;
1151         }
1152     //Hinv is overwitten with the inverse...
1153     inverse(H,npar,Hinv);
1154 
1155 /*****************************************************************/
1156 
1157     cudaMemcpy(d_theta_new,theta_new,npar*sizeof(double),cudaMemcpyHostToDevice);
1158     cudaMemcpy(d_Hinv, Hinv, npar*npar*sizeof(double),cudaMemcpyHostToDevice);
1159 
1160     cudaStat = cudaDeviceSynchronize();
1161 
1162     updateThetaTrace<<<1,1>>>(d_1theta, d_theta_new, d_trace, npar, d_Hinv);
1163     cudaStat = cudaDeviceSynchronize();
1164 
1165     cudaMemcpy(theta, d_1theta,npar*sizeof(double),cudaMemcpyDeviceToHost);
1166     cudaMemcpy(trace,d_trace, sizeof(double),cudaMemcpyDeviceToHost);
1167 
1168     cudaStat = cudaDeviceSynchronize();
1169 
1170 
1171 /*****************************************************************/
1172 
1173     cudaMemcpy(d_Ew_new, Ew_new, sizeof(double),cudaMemcpyHostToDevice);
1174     cudaMemcpy(d_Ed_new, Ed_new, sizeof(double),cudaMemcpyHostToDevice);
1175 
1176     cudaStat = cudaDeviceSynchronize();
1177 
1178     lastblock<<<1,1>>>(d_gamma, d_alpha, d_beta, d_Ew_new, d_Ed_new, d_trace, n, npar,
         d_epoch);
1179     cudaStat = cudaDeviceSynchronize();
1180 
1181     cudaMemcpy(epoch, d_epoch, sizeof(int), cudaMemcpyDeviceToHost);
1182     cudaMemcpy(alpha, d_alpha, sizeof(double), cudaMemcpyDeviceToHost);
1183     cudaMemcpy(beta, d_beta, sizeof(double), cudaMemcpyDeviceToHost);
1184     cudaMemcpy(gamma, d_gamma, sizeof(double), cudaMemcpyDeviceToHost);
1185     cudaStat = cudaDeviceSynchronize();
1186 
1187 /*****************************************************************/
1188 
1189   } //end main while
1190 
1191 
1192     cudaFree(d_1X);
1193     cudaFree(d_1theta);
1194     cudaFree(d_y);
1195     cudaFree(d_yhat);
1196     cudaFree(d_residuals);
1197     cudaFree(d_Ed);
1198     cudaFree(d_Ew);
1199     cudaFree(d_alpha);
1200     cudaFree(d_beta);
1201     cudaFree(d_gamma);
1202     cudaFree(d_1J);
```

```
1203     cudaFree(d_H);
1204     cudaFree(d_Hinv);
1205     cudaFree(d_diag_H);
1206     cudaFree(d_g);
1207     cudaFree(d_C);
1208     cudaFree(d_flag_C);
1209     cudaFree(d_theta_new);
1210     cudaFree(d_trace);
1211     cudaFree(d_Ew_new);
1212     cudaFree(d_Ed_new);
1213     cudaFree(d_epoch);
1214 
1215     free(X);
1216     free(y);
1217     free(yhat);
1218     free(residuals);
1219     free(J);
1220     free(H);
1221     free(Hinv);
1222     free(theta);
1223     free(theta_new);
1224     free(delta);
1225     free(g);
1226     free(diag_H);
1227     free(F_history);
1228 
1229     free(gamma);
1230     free(Ed);
1231     free(Ed_new);
1232     free(Ew);
1233     free(Ew_new);
1234     free(alpha);
1235     free(beta);
1236     free(C);
1237     free(C_new);
1238 
1239     free(epoch);
1240   free(max_epochs);
1241     free(mu);
1242     free(mu_inc);
1243     free(mu_dec);
1244     free(mu_max);
1245     free(change);
1246     free(trace);
1247     free(flag_C);
1248 
1249     cudaDeviceReset();
1250 
1251   return 0;
1252 }
```

Anexo C: Instrucciones para la compilación del programa *trainbr.cu*

El siguiente comando realiza la compilación del programa para la regularización Bayesiana de una red neuronal artificial en el sistema operativo Linux.

```
#! /bin/bash

# se debe compilar para la arquitectura sm_35 o superior para este trabajo
# se hizo para la arquitectura sm_61 que es la arquitectura específica de
# la GPU utilizada

# se genera primero el código objeto

nvcc -arch=sm_61 -o trainbr.o trainbr.cu

# se realiza el enlace de las bibliotecas necesarias (linking)
# se deben incluir las bibliotecas cublas y cusolver en caso de que no esten
# en la ruta se agregan a la variable de entorno PATH


nvcc -arch=sm_61 -o trainbr trainbr.o -lcublas -lcusolver
```